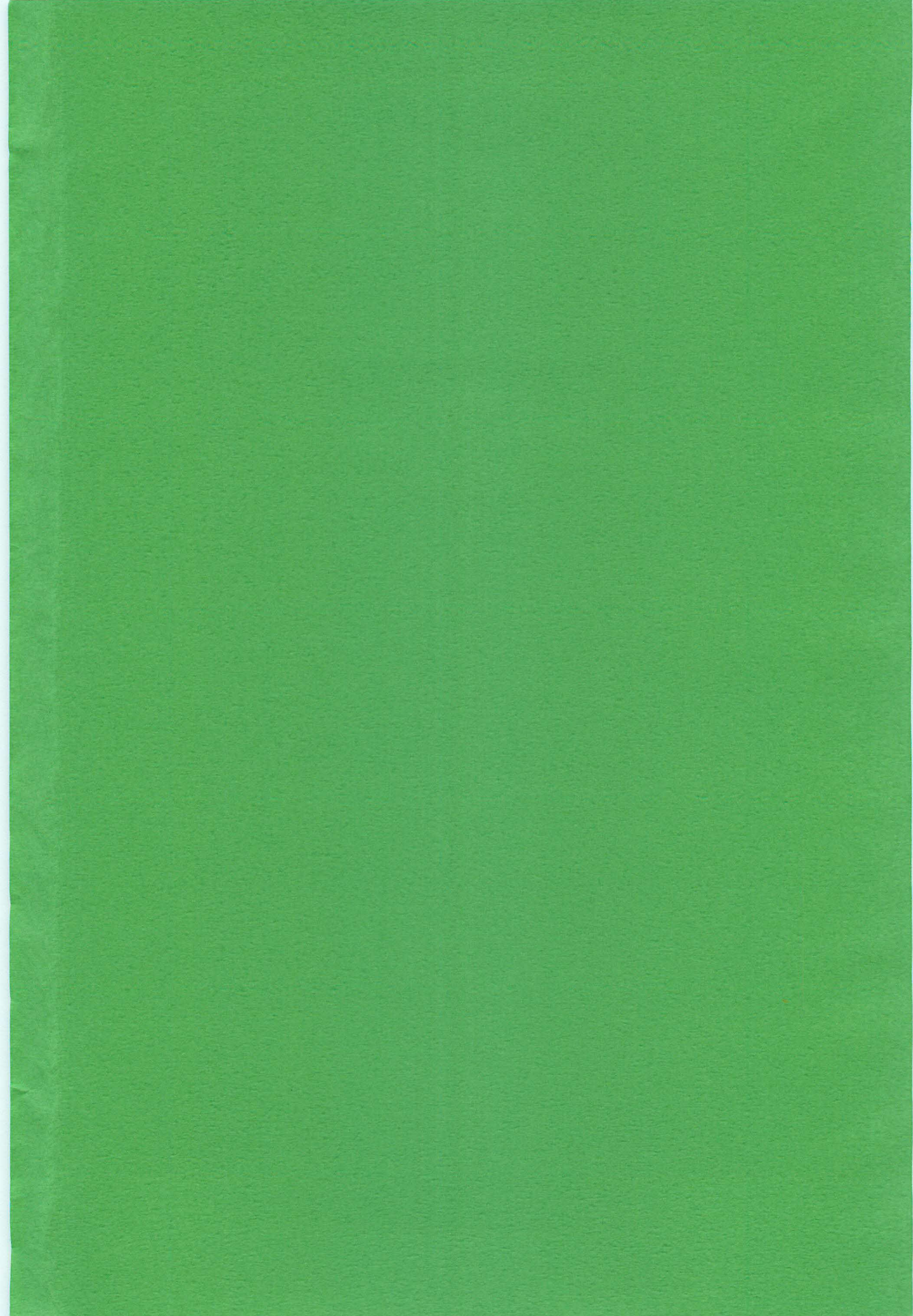
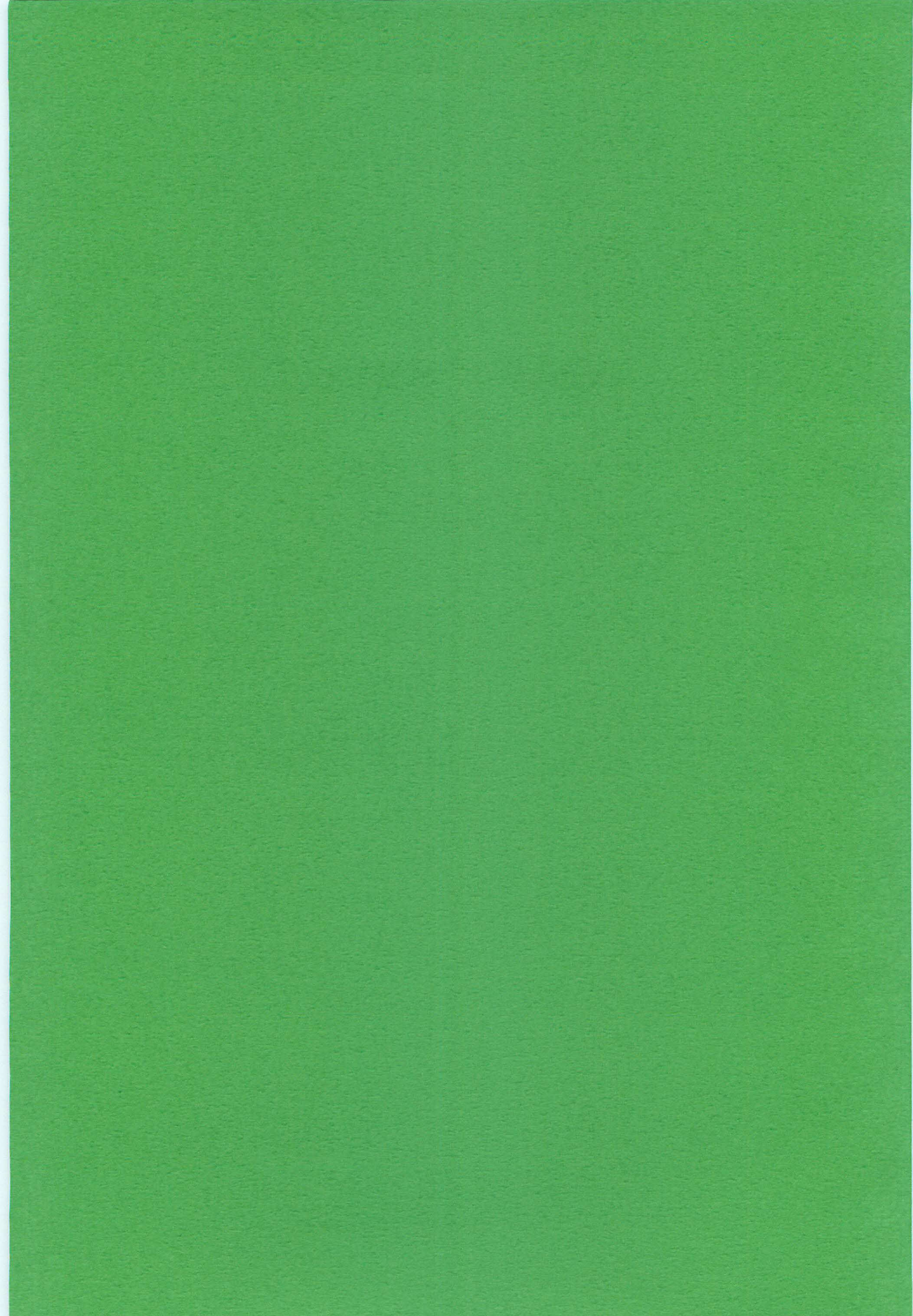
# User's Reference

Manual
Books

# X68k

Programming Series

村上敬一郎＋萩野祐二＋大西恵司…………共著

(#2)

# X680x0 libc

## User's Reference

*LIBC*は **X68000** および **X68030** 上の **Human68k** ver.2，**Human68k** ver.3 上で動作しますが，**GCC** あるいは *XC*がすでに動作していることが必要です。したがって，**GCC** を利用する場合には最低限 2M バイトのメモリが必要です。

**本書は,「*X680x0 Develop. & libc II*」の内容に即して「*X680x0 libc*」を加筆修正したものです。**

**バージョンアップによって仕様変更されたライブラリ関数については，新規に作成した「変更」項目にその内容が解説してあります。**

- システム名，CPU 名などは一般に各社の登録商標です。本文中では，とくに TM，Ⓡは明記しておりません。

# はじめに

「*X68k Programming Series #2 X680x0 libc*」は，**Human68k** と **UNIX** という OS の違いを吸収し，他の処理系と **X68000** の間でプログラムの移植性を高めるために作成されたライブラリです。また，**SHARP** 純正の *XC* との互換性を考慮して *XC* コンパチブルヘッダも付属していますから，*XC* ライブラリを前提として作成されたプログラムとも，ソースレベルで互換性を保つことができます。そのほかにも，ユーザの環境に合わせて数値演算コプロセッサを直接駆動することができる数学関数や **UNIX** ライクなシグナル機構など，**X68000** のもつ機能をこれまでよりもいっそう発揮できるような関数群を備えています。

*LIBC* は，「*X68k Programming Series*」の「*#1 X68000 Develop.*」で提供されている開発ツールを使用して作成されています。そのため，これらのツールと合わせて使用することによって，より高度なプログラム開発を行うことができるようになるでしょう。また，ライブラリの全ソースコードが公開されているだけではなく，ほとんどの関数が C 言語によって記述されていますから，ユーザの皆さん自身の手でライブラリのメンテナンス・カスタマイズ・機能追加などを容易に行うことができます。

*LIBC* が **X68000** および **X68030** の世界を広げることに少しでも役立つことができれば幸いです。

1993 年 3 月

***Project LIBC Group***

# Manual Books 発刊へのまえがき

前著「*X68k Programming Series #2 X680x0 libc*」を発行したのが 1993 年 5 月のことでしたから，それからちょうど 1 年近く経ったでしょうか。最初に発表することができた ***LIBC***バージョン 1.0.20 も，その後，改版を続けて 1994 年 7 月現在ではバージョン 1.1.31 にまでいたっています。

フリーウェア (事実上は PDS) としてソースコードまですべて公開したこともあり，思った以上にさまざまな反響をいただきました。不具合の修正や報告を，パソコン通信あるいは各種のネットワークを通じていただいたことも数多くあります。また，新たな追加機能を contribute していただいたこともありました。今や ***LIBC***は，これら多くの方々の協力がすべて結集したものになっているといえるかもしれません。

さて，本書「*X680x0 libc Manual Books*」を作成するにあたり，旧版から変更された部分などの相違については，「*X680x0 Develop. & libc II*」で紹介した内容を完全に反映させるよう心がけました。したがって，マニュアル版ではかなり今現在の ***LIBC***の仕様に忠実なものとなっています。しかし ***LIBC***はこれからも成長していくでしょうし，提供する関数の数も増えていくでしょう。その意味では，マニュアルと実物とで仕様が異なる関数も出てくると予想されますが，それについてはあらかじめご了承していただきたく思います。

最後に，この場をお借りしてあらためて協力してくださった多くの，実に多くの方々にお礼申し上げます。**ANSI C** 対応をうたってはいるものの，正直いってどちらかというと **UNIX** 寄りである「特殊な」ライブラリにもかかわらず支持してくださっている皆さん，不具合を報告/修正していただいた皆さん，そして新たなソースコードを送ってくださった皆さん，どうもありがとうございました。

1994 年 7 月

***Project LIBC Group***

X68k Programing Series #2

# X680x0 libc

Vol. 1

User's Reference

# C O N T E N T S

Chapter 1


*LIBC*リファレンス …… 1

1.1 ―― インストール …… 2
- インストールの準備 …… 2
- インストールする …… 3
- ライブラリパッケージのインストール …… 5
- ソースコードパッケージのインストール …… 10

1.2 ―― 使い始める前に …… 14
- *LIBC*について …… 14
- *LIBC*は PDS である …… 15
- *LIBC*と *XC* …… 17

1.3 ―― インクルードファイルの使い方 …… 21
- インクルードファイルの互換性 …… 21
- インクルードファイル一覧 …… 22
- インクルードファイルの使い方 …… 27
- インクルードファイルについて …… 28

1.4 ―― ライブラリファイルの使い方 …… 31
- ライブラリファイル一覧 …… 31
- ライブラリファイルの使い方 …… 37
- ライブラリファイルについて …… 39


Chapter 2


*LIBC*プログラミング …… 41

2.1 ―― ファイル …… 42
- ファイルを調べる …… 42
- ファイルの構造 …… 48
- パス名 …… 50

2.2 ―― 浮動小数点演算 …… 51
- 数値の表現形式 …… 51
- 数学関数の使用について …… 54
- コプロセッサ補足説明 …… 57

2.3 ―― 日付と時間 …… 61
- 暦時間 …… 61
- 協定世界時 …… 64
- 地域時間 …… 64
- 詳細時間 …… 67

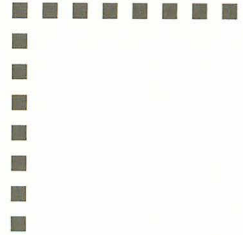
2.4 ―― ヒープ領域 …… 72
- ヒープ領域の実際 …… 72
- 配列と動的メモリ確保 …… 75
- ヒープ領域の使い方 …… 79

2.5 ——— **スタック領域** ……… 83
● スタック領域の使われ方 ……… 83
● 可変長引数 ……… 88
2.6 ——— **シグナル機構** ……… 92
● シグナル機構について ……… 92

## Chapter 3

**Appendix A** ……… 97

A ——— ***LIBC*の起動オプション** ……… 98
B ——— ***LIBC*のエラーメッセージ** ……… 100
C ——— ***LIBC*のエラーコード** ……… 101
D ——— ***LIBC*が見る環境変数** ……… 103
E ——— ***LIBC*とフリーウェア** ……… 106
F ——— ***LIBC*用語一覧** ……… 108

## Chapter 4

**Appendix B** ……… 113

A ——— **システムのエラーコード** ……… 114
B ——— **Human68k の内部情報** ……… 116

**参考文献** ……… 130

**索　　引** ……… 132

# Chapter 1
# *LIBC*リファレンス

　実際にライブラリを使い始める前に，*LIBC*について少し説明しておきたいと思います。*LIBC*は *XC*に代わるものではありませんし，すべての場合に役に立つわけではありません。*LIBC*と *XC*の関係やそれぞれの得手不得手を知っておいてください。
　また，*LIBC*のインストール手順やインクルードファイル，ライブラリファイルの使い方についても解説します。

# 1.1 ………… インストール

## ❖ インストールの準備

***LIBC***はパソコン通信上などでパッケージとして配布されています。これらのパッケージにはそれぞれインストールプログラムが付属しています。ここではそのインストールプログラムの使い方を解説します。実際に***LIBC***を入手して，インストールする際に参考にしてください。

なお一例として，***LIBC***はパソコン通信NIFTY-ServeのSHARP Users' フォーラム・ワークステーション館 (FSHARP3) のデータライブラリに登録されています。

実際にインストールをする前に，ディスクと環境の準備が必要です。説明をよく読んで，インストールに備えてください。

### ◆ ディスクの準備

***LIBC***は2つのパッケージで構成されています。1つはライブラリパッケージで，実際に***LIBC***を利用するためのインクルードファイルやライブラリファイルが含まれます。もう1つはソースコードパッケージで，***LIBC***のすべてのソースコード，付属資料などライブラリパッケージに含まれないものすべてが入っています。

これら2つのパッケージのうちライブラリパッケージは必ず必要ですが，ソースコードパッケージはサイズが非常に大きいため，インストールするかどうかはあなたが選択できるようになっています。それぞれのパッケージは概算でTable 1-1に示されるだけのディスク容量を必要とします。

Table 1-1で作業領域とあるのは，インストール時に一時的に使用するディスク領域です。インストール完了後はこの作業領域は必要なくなり，インストール容量で示された容量だけを占有します。したがってライブラリパッケージをインストールするには，最低限2Mバイトのメモリと約1Mバイトのディスク容量が必要になります。これはフロッピーディスクでも不可能ではない大きさですが，

**Table 1-1** ● インストールに必要なディスク容量

| パッケージ名 | 作業領域 | インストール容量 |
|---|---|---|
| ライブラリパッケージ | 1Mバイト | 1Mバイト |
| ソースコードパッケージ | 4Mバイト | 3.5Mバイト |

できればハードディスクが望ましいでしょう。もちろんソースコードパッケージのインストールには，ハードディスクが必ず必要となります。

もし残りディスク容量がここで示した容量に満たない場合は，インストールプログラムが警告を表示して処理を中断します。

### ◆ 環境の準備

*LIBC*のインストールを行う際には，特別環境を設定しなおす必要はありません。**GCC** なり *XC*が動作するだけの環境[1]が整っていればかまいません。

1)環境設定については，GCCや *XC*のマニュアルを参照してください。

ただし，ソースコードパッケージをインストールする場合は，環境を整える必要があります。ソースコードパッケージ中のファイルのなかには，非常に長いファイル名のものもあるので，そのままインストールするといくつかのファイルが重複して消えてしまいます。これを防ぐためには，ソースコードパッケージをインストールする前に，ファイル名を 21 文字すべて認識させるよう，環境を設定しなければなりません。

もしあなたが **Human68k** ver.2，ver.3 を使っているならば，“`TwentyOne.X`” というフリーウェアが利用できます。しかし，**Human68k** ver.1 を使っている場合は，残念ながらいくつかのファイルが失われる可能性があることを黙認しなければなりません[2]。“`TwentyOne.X`” の設定方法やインストールについては，*LIBC*のインストールの説明でもう一度触れますので，このまま読み進んでください。

2)したがって，自分で *LIBC* のライブラリを再構築することはできません。

## ❖ インストールする

準備はもう整ったでしょうか。それではいよいよインストール作業に入ります。説明をよく読んで，まちがいのないように気をつけてください。

### ◆ 起動する

まず **X68000** なり **X68030** の電源を入れ，あなたが普段使っている環境で立ちあげます。フロッピーディスクだけで使用している人は，そのシステムディスクから，またハードディスクを使っている人はそのハードディスクから **Human68k** を起動してください。また，あなたが **SX–Window** を使っているのならば，**SX–Window** を終了させてコマンドラインに戻るか，“`COMMAND.X`” のアイコンをダブルクリックしてコマンドシェルを起動してください。

次に，ダウンロードしたアーカイブを納めたディレクトリにカレントディレクトリを変更してください。たとえば “`C:\ARCHIVE`” ディレクトリにパッケージファイルを格納してあれば，次のようにタイプします。画面の表示例中の “`A:\>`” という部分は **Human68k** のプロンプトです。

```
A:\> C:\ARCHIVE
```

最後に，そこから“SETUP.X”を起動します。次のようにタイプしてください。

```
C:\> SETUP.X
```

### ◆ インストールを始める

正しく“SETUP.X”が実行されると，画面にインストールプログラムのメッセージが表示されます。基本的なインストールの方法は非常に簡単なもので，画面に表示された質問に答えていくだけです。以下，順を追って説明していきます。質問の意味をよく理解して，正しく答えるようにしてください。

なお，画面の表示例中のCTRL+Cとは，CTRLキーを押しながらCを押すことを表します。

```
*************************************************************
*      LIBC - Project C Library Group - Install Script      *
*************************************************************

今から LIBC をインストールします。質問に正しく答えてください。

(質問)
LIBC のパッケージのうち，どれをインストールしますか?
  <1> ライブラリパッケージ     (作業用に 1MB 必要です)
  <2> ソースコードパッケージ (作業用に 4MB 必要です)
  <3> 両方のパッケージ         (作業用に 5MB 必要です)
1,2,3 のどれかを選択してください。

中断するには CTRL+C を押してください。

(回答) --->
```

まずインストールするパッケージを選択してください。この選択をする前にハードディスクのどのディレクトリにインストールするか，またそのディレクトリの空き容量が十分にあるかどうかについて調べておいてください。

それでは<1>番を選んだ方は次の「ライブラリパッケージのインストール」(P.5)を，<2>番を選んだ方はその次の「ソースコードパッケージのインストール」(P.10)を，<3>番を選んだ方は両方を読み進んでください。両方を選んだ場合は，まずソースコードパッケージのインストールから始まります。

## ❖ ライブラリパッケージのインストール

本セクションではインストールプログラムのメッセージにしたがって，*LIBC*のライブラリパッケージをインストールする手順について説明します。

### ◆ インストール先の指定

まずインストールするディレクトリを指定してください。インストールプログラムは指定されたディレクトリにすべてのデータをインストールできるかどうか，空き容量をチェックし，十分な空きがあればインストールを開始します。

```
(質問)
それでは「ライブラリパッケージ」をインストールします。
どこにインストールしますか? ディレクトリ名で答えてください。

中断するには CTRL+C を押してください。

(回答) --->
```

たとえば“A:”ドライブの“\usr\lang”というディレクトリにインストールしたいときは，次のようにタイプしてください。

```
(回答) ---> A:\usr\lang
```

ここで，もしディスクの空き容量が足りない場合は次のようなメッセージが表示され，インストールプログラムは中断します。インストールするディレクトリをもう一度確かめてから，インストールを最初からやり直してください。

```
(警告)
指定したディレクトリには「ライブラリパッケージ」を
インストールするだけの空き容量がありません。

もう一度確認して最初からインストールをやり直してください。

C:\>
```

ディスクの空き容量の確認が終わると，インストールプログラムはパッケージのインストールを開始します。約2，3分かかりますから，お茶でも飲みながらしばらく作業を見守っていてください。

```
それでは「ライブラリパッケージ」をインストールします。

include/
include/a_out.h
include/alloca.h
include/assert.h
   :
   :
 中略
   :
   :
README
WARNING

インストールは無事終了しました。

C:\>
```

### ◆ ライブラリ形式の選択

*LIBC*の標準構成では，すべてのライブラリが“.a”形式に統一されています。もしあなたが*XC*に付属している“LIB.X”をもっているならば，これらをより高速な *XC* ver.2 の“.l”形式に変換することができます。

**“.l”形式のライブラリのほうが高速にリンクできる**

```
(質問)
ライブラリの形式を高速な .l 形式に変換しますか?
  変換するなら "y" を
  変換しないなら "n" を
選択してください。

中断するには CTRL+C を押してください。

(回答) --->
```

“y”を選択すると，インストールプログラムはあなたの環境のなかから“LIB.X”を探し，それを使ってライブラリを“.l”形式に変換します。もし“LIB.X”が見つからなければ，インストールプログラムは次のようなメッセージを表示し，名称の変更を行いません。

```
(警告)
LIB.X が見つかりません。
ライブラリの形式は .a 形式のままです。
```

正しく“LIB.X”が見つかった場合，インストールプログラムはすべてのライブラリを“.l”形式に変換します。

```
ライブラリの形式を .l 形式に変換します。

変換 libc.a --> libc.l
変換 libdos.a --> libdos.l
   :
   :
 中略
   :
   :
変換 libw.a --> libw.l

変換は無事終了しました。
```

### ◆ ライブラリ名の選択

*LIBC*の標準構成ではライブラリの名称が“libc.a”というように，*XC*とは前後が逆になっています[4]。GCC コンパイラと *LIBC*ライブラリという組み合わせだけで使用するならばこのままでかまいませんが，*XC*コンパイラと *LIBC*ライブラリの組み合わせで使用したり，*XC*環境と併用したい場合は名称を *XC*形式に直さなければなりません。

4)詳しくは「ライブラリファイルの使い方」(P.37)を読んでください。

```
(質問)
ライブラリの名称を XC 形式に変更しますか?
  "clib.a" という XC 形式のライブラリ名を使用するなら "y" を
  "libc.a" という LIBC 形式のライブラリ名を使用するなら "n" を
選択してください。

中断するには CTRL+C を押してください。

(回答) --->
```

もし *XC*形式に修正したければ上記の質問に“y”で，標準構成のままでよければ“n”で答えてください。なお，ライブラリの形式を“.l”に変更すると選択した場合，これ以降の画面の表示は“.a”ではなく，“.l”になります。文中の表示例を，適宜読み変えてください。なお，バージョンアップなどで ***Project LIBC Group***が *LIBC*を再配布する場合は，つねに“.a”形式で行いますから，なるべくならば“.a”形式のまま使用するようにしてください。

```
(回答) ---> y

ライブラリの名称を XC 形式に変更します。

変更 libc.a --> clib.a
変更 libdos.a --> doslib.a
   :
   :
 中略
   :
   :
変更 libw.a --> wlib.a

変更は無事終了しました。
```

### ◆ AUTOEXEC.BATの書き換え

ここまででライブラリパッケージのインストールは終わりました。後はあなたの“AUTOEXEC.BAT”を書き換えて，*LIBC*が正しく動作するように，環境を設定するだけです。環境設定は自動または手動で行うことができます。

```
(質問)
最後にここまでのインストール結果に合わせて環境設定を行います。
環境変数の設定 (AUTOEXEC.BAT の書き換え) を自動で行いますか?
  自動で行いたい場合は "y" を
  自分で行いたい場合は "n" を
選択してください。

中断するには CTRL+C を押してください。

(回答) --->
```

もしあなたがコマンドシェルとして“COMMAND.X”を使用していて，自分の“AUTOEXEC.BAT”を「自動」で書き換えてほしいときには，上記の質問に“y”と答えてください。次に“AUTOEXEC.BAT”のあるドライブを聞いてきますから，その質問にも答えてください。ドライブ名には“:”をつけてもつけなくてもかまいません。

これで，必要な設定があなたの“AUTOEXEC.BAT”の「最後」に自動的に追加されます[5)]。次の画面表示例ではパッケージを“A:\usr\lang”にインストールし，ライブラリの形式に“.l”を，またライブラリの名称は変更しなかった場合を想定しています。

5)後は必要に応じて，エディタなどで編集して使用してください。もちろん編集しなくても使えます。

```
(回答) ---> y

AUTOEXEC.BAT を自動的に編集します。

(質問)
AUTOEXEC.BAT はどのドライブにありますか?
ドライブ名を, たとえば A: というように答えてください。

(回答) ---> A:

AUTOEXEC.BAT の最後に以下の行を追加します。

  set include=A:\usr\lang\include
  set lib=A:\usr\lang\lib
  set GCC_LIB=.l
  set GCC_NO_XCLIB=yes

インストールはこれですべて終わりです。
おつかれさまでした。

C:\>
```

反対に，勝手に“AUTOEXEC.BAT”を書き換えられては困るという場合やコマンドシェルとして“COMMAND.X”以外のプログラムを使っている人は，環境設定を手動で行ってください。手動で環境設定を行う場合，必要な環境設定が画面上に表示されます。この環境設定をメモし，後はエディタなどを使い，自分の好みの環境にしてください。

表示される形式は“COMMAND.X”に対するものなので，もしコマンドシェルにksh.xやfish.xなどのプログラムを使用している場合はそのままでは使えません。適切な形に直してから編集してください。

```
(回答) ---> n

環境設定は手動で行ってください。
具体的には AUTOEXEC.BAT に次の行を追加するか,
あるいはすでにある行を変更してください。

  set include=A:\usr\lang\include
  set lib=A:\usr\lang\lib
  set GCC_LIB=.l
  set GCC_NO_XCLIB=yes

インストールはこれですべて終わりです。
おつかれさまでした。

C:\>
```

## ❖ ソースコードパッケージのインストール

本セクションではインストールプログラムのメッセージにしたがって，*LIBC* のソースコードパッケージをインストールする手順について説明します。

### ◆ 環境のチェック

「環境の準備」(P.3) でも述べましたが，ソースコードパッケージを正しくインストールするためには，ファイル名を 21 文字まで正しく認識させるよう，あらかじめ環境を設定しなくてはなりません。

**Human68k** はファイル名を 21 文字まで扱えるにも関わらず，標準設定ではそのうち先頭の 8 文字までしか認識しません。そのため，9 文字目以降が異なっていても先頭さえ同じなら，別のファイルを同じファイルだと認識してしまいます。ソースコードパッケージ中のファイルは非常に長いファイル名が多いので，このままではいくつかのファイルの名前が重複してしまいます。

もしあなたが **Human68k** ver.2，ver.3 を使っているならば，“`TwentyOne.X`” というフリーウェアを利用することで，この問題を解決することができます。しかし，**Human68k** ver.1 を使っている場合，現段階では残念ながらあきらめざるを得ません。いくつかのファイルが失われることを覚悟してください。

#### ファイル名を 21 文字すべて認識させること

まず，インストールプログラムはあなたの **Human68k** のバージョンをチェックし，それに応じていくつかの質問をしてきます。

### ◆ Human68k ver.2 / ver.3

**Human68k** ver.2 あるいは ver.3 を使っている場合，インストールプログラムはあなたが “`TwentyOne.X`” をすでに使っているかどうかを調べ，もし使用していなければ次のようなメッセージを表示してきます。

```
(質問)
あなたは TwentyOne.X を常駐させていないようです。
ソースコードパッケージを正しくインストールするには
このプログラムを常駐させておかなければなりません。

TwentyOne.X をどのディレクトリにコピーしますか。
ディレクトリ名で答えてください。

中断するには CTRL+C を押してください。


(回答) --->
```

“TwentyOne.X” をインストールするディレクトリを指定してください。たとえば “A:\usr\bin” にコピーしたければ，次のようにタイプします。

```
(回答) ---> A:\usr\bin
```

ディレクトリを指定すると，インストールプログラムはそこに “TwentyOne.X” をコピーし，次のようなメッセージを表示して処理を中断します。そのメッセージにしたがって自分で “CONFIG.SYS” か “AUTOEXEC.BAT” を書き換え，**X68000** を再起動 (リセット) してください。

```
(回答) ---> A:\usr\bin

あなたの CONFIG.SYS に次の 1 行を書き加えてから
リセットボタンを押してください。
そして X68000/X68030 が正しく起動したら
もう一度最初からインストールを行ってください。

  DEVICE = A:\usr\bin\TwentyOne.X +TS

あるいは，あなたの AUTOEXEC.BAT に次の 1 行を書き加えてから
リセットボタンを押してください。
そして X68000/X68030 が正しく起動したら
もう一度最初からインストールを行ってください。

  A:\usr\bin\TwentyOne.X +TS

C:\>
```

### ◆ Human68k ver.1

**Human68k** ver.1 を使っている場合，インストールプログラムはファイルが失われる可能性があることを警告します。

```
(警告)
HUMAN ver.1 では今のところ，ファイル名を 21 文字すべて
認識させることができません。
結果として，いくつかのファイルが失われることになります。
あらかじめ了解しておいてください。
```

### ◆ インストール先の指定

ソースコードをインストールするディレクトリを指定してください。インストールプログラムは，指定されたディレクトリにすべてのデータをインストールできるかどうか，空き容量をチェックし，十分な空きがあればインストールを開始します。

```
(質問)
それでは「ソースコードパッケージ」をインストールします。
どこにインストールしますか?
ディレクトリ名で答えてください。

中断するには CTRL+C を押してください。

(回答) --->
```

たとえば“A:”ドライブの“\usr\lang”というディレクトリにインストールしたいときは，次のようにタイプしてください。なお，ライブラリパッケージとソースコードパッケージは別々のディレクトリにインストールすることもできます。しかし，もし可能であれば，できるだけ同じディレクトリにインストールするようにしてください。

```
(回答) ---> A:\usr\lang
```

ここで，もしディスクの空き容量が足りない場合は次のようなメッセージが表示され，インストールプログラムは中断します。インストールするディレクトリをもう一度確かめてから，インストールを最初からやり直してください。

```
(警告)
指定したディレクトリには「ソースコードパッケージ」を
インストールするだけの空き容量がありません。
```

ディスクの空き容量の確認が終わると，インストールプログラムはパッケージのインストールを開始します。場合にもよりますが，すべてを展開するにはおよそ10分から15分かかります。

ソースコードパッケージのインストールはこれで終了です。「ライブラリパッケージ」もいっしょにインストールすると指定した場合は，この次にライブラリパッケージのインストールが開始されます。前節の「ライブラリパッケージのインストール」(P.5) へ戻ってください。

```
それでは「ソースコードパッケージ」をインストールします。

src/
src/DefaultBases
src/DefaultRules
src/Makefile
    :
    :
  中略
    :
    :
README
WARNING

インストールは無事終了しました。
おつかれさまでした。

C:\>
```

# 1.2 ……………………使い始める前に

## ❖ *LIBC*について

*LIBC*を使い始める前に，*LIBC*の生い立ちについて説明しておきます。まず，*LIBC*は決して機能や性能という側面からではなく，あくまでも**フリー**(自由)な開発環境を提供するために，作成されたものだということを認識しておいてください。

### ◆ *LIBC*の意味

*LIBC*の作成は**X68000**の**Human68k**上で使うことのできる，まったくフリー(自由)な[1]ライブラリという位置付けで，1991年に開始しました。確かに*XC*はソースコードが最初から付属していましたし，使用にあたってもほとんど制限はありません。そういう意味では，*XC*はバグが多かったとはいえ，素晴らしいものには違いありませんでした。

1)著作権の制限を受けないという意味でフリーな。

しかし**Human68k**上の開発環境は，すでにその当時，*XC*が発売されたころとはすっかり様子が変わっていました。**GCC**(*GNU C Compiler*)によってCコンパイラが，また**HAS**(*High-speed Assembler*)，**HLK**(*High-speed Linker*)，**HAR**(*High-speed Archiver*)によってアセンブラ，リンカ，アーカイバが，そして**GDB**(*GNU Debugger*)によってソースコードデバッガまでがフリーウェアとして手に入るようになっていたのです。そのような状況下で，最後までフリーではなかったのはライブラリだけで，逆にいえばライブラリがフリーになるのは，ある意味では必要なことでした。

### ◆ 開発の開始と中止

*LIBC*は，最初はいろいろな人に好きな関数を作ってもらい，それをまとめる形で完成させる予定でした。実際，商用BBSである`Nifty-Serve`内にホームパーティー[2]を設置し，そこで開発していましたが，それは次のような問題によって中止せざるを得ませんでした。

2)月額料金で開くことができ，趣向の同じ人どうしが集まることのできる小さな会議室。

1. **重複**

   まず，各人から好きな関数を寄せ集めても，その大半が重複してしまうことが問題でした。それも，特に一部の関数に集中して重複してしまったのです。

そういった重複した関数ばかりが集まる一方で，それ以外の，本当に重要な部分は何もできあがりそうにありませんでした。

**2. 混乱**

ライブラリはさまざまな規格によって仕様がきちんと定まっています。しかし，どのような規格をベースとして採用するかは，依然として流動的でしたし，規格に正しく合致しているかを調べるのは大仕事でした。また，ライブラリ内の関数は他のいろいろな関数と密接に結びついているため，それらの間のインタフェイスを取らねばならないことも難問でした。

**3. 障害**

もともとが有志による活動だったせいもありますが，ライブラリ自体の作成は遅々として進みませんでした。また著者自身の仕事などの都合もあり，言いだしっぺ本人が作成作業を中断してしまいました。

結局のところ，自由気ままにやっていてできるようなものではないという現実だけがわかりました。こうしてライブラリの作成という企画は，棚にあがったまま4か月近くも放置されるはめになったのです。もしそのまま何もなければ，永久に消え去っていたかもしれません。

◆ 開発の再開

この企画が再度もちあがったのは，実はソフトバンクのおかげでした。1992年の3月に，ライブラリを題材にして本を出してみようという話になったのです。今このライブラリがあるのは，すべてこのときの担当者の方々の決断によるといっても過言ではないかもしれません。

再開されたこの企画は*LIBC*と名付けられました。これは**UNIX**上でC言語の標準ライブラリである“`libc.a`”の名前から取ったものです。普通，libcといえば標準ライブラリのことを指しますが，本書ではこのような経緯から，*LIBC*をライブラリ全体[3)]の総称として使っています。

3)つまり，標準関数ではないものも含めてすべて。

## ❖ *LIBC*はPDSである

*LIBC*はフリー(自由)な開発環境を支援するために，PDSとして公開されています。次に述べる事柄をよく理解し，その趣旨を理解していただければ***Project LIBC Group***としてもうれしいかぎりです。

◆ PDSとは

PDSとは*Public Domain Software*の略語で，簡単にいえば「著作権が放棄されたもの。みんなのもの(*Public Domain*)」的な意味あいをもっています。著者は法律には詳しくないので，あまり詳しいことはいえませんが，たとえばアメリカでは著作物に明確に“`Copyright`”と記さないと著作権が発生しないそうです。そ

こでPDSでは，故意にこの“Copyright”を書かないことで著作権を放棄し，誰にでも自由に使ってもらおうというわけです。

一方，これに対して**フリーウェア**や**シェアウェア**というものがあります。フリーウェアとは誰にでも使ってもらおうという点ではPDSと同じですが，著作権は明確に宣言されており，ソフトウェア作者の意志でどこまで自由に扱えるかも異なっています。また，シェアウェアはフリーウェアと似ていますが，気にいったらお金を送ってほしいというただし書きがつきます。

以前，おそらく著者の思うところでは5，6年以上前ではPDSというのはわりと一般的に存在していたと思うのですが，最近はすっかり見かけなくなりました。今は，特に日本では，ほとんどすべてがフリーウェアかシェアウェアという形態をとっているようです[4)]。というのも，どうやら日本での著作権の扱い方があちらとは異なっているからのようでした。

4)しかしなかには筆者のように，PDSにこだわる人間もいます。

日本ではアメリカとは異なり，特に明示的に“Copyright”と書かなくても自動的に著作権が発生するそうです。しかも，本人が希望してもどうしても放棄できない著作権があるということで，つまり正確にいえば，日本ではPDSという形態そのものが成立しないのだそうです。

### ◆ なぜ *LIBC* は PDS か

フリーウェアやシェアウェアは誰でも自由に使える[5)]側面をもっていますが，やはり著作権によって保護されており，使用に制限が課せられています。

5)これは作者の意志によって制限が加えられる場合があります。

著作権者の立場で考えればこれは当然のことであり，著作権は厳格に守られねばなりません。しかし逆に使う側の立場からすれば，たとえ「誰でも自由に使ってください」とはいえ，「使わせてもらう」立場には違いありません。また，そのソフトウェア中の一部を参考にしたり，流用したりするには著作権者の了解が必要です[6)]。これでは自由なソフトウェアを開発するのに，少なからず障害になるのではと筆者は考えました。そこで，使う側の人が何の気兼ねもなく自由にソースコードを利用したり，参考にしたりできるように*LIBC*をPDSとして公開することにしたのです。

6)こういうことが無条件に認められている場合もありますが，一般にはちゃんと了解を得るのが礼儀です。

### ◆ *LIBC* は PDS である

以上のことを踏まえ，我々は*LIBC*のライブラリ，オブジェクト，インクルードファイルからそのソースコードにいたるまで，そのすべてをPDSとして公開します。誰でも自由に*LIBC*を使えるし，コピーすることも，改変することも，また自分のソフトウェアに組み込むことさえできます。*Project LIBC Group*にはソースコードを書く人も，テストする人も，そして保守する人もいますが，それは単にそれだけの存在であり，著作権者という立場ではありません。

我々は*LIBC*に関するすべての著作権を放棄します。また放棄できない部分についても，*LIBC*が意味的にPDSとして成立するよう，著作権を主張しないことを宣言します。我々が知識の自由な共有のためにできることはこのくらいしかありませんが，*LIBC*がすべての人の共有財産として，何かの役に立てばと考えています。

ただ，以下のことには注意してください（いささか矛盾ですが…）。

**LIBCはPDSですが，本書はPDSではありません**

**◆ コントリビューション**

*LIBC*へのコントリビューション[7]はいつでも歓迎します。著作権を放棄し，前述したような*LIBC*のポリシーにしたがっていただけるなら，あなたが作成した関数を我々に提供してください。我々は，それらコントリビューションしていただいたソースコードを，将来のバージョンで*LIBC*に統合，配布したいと思っています。

7)コントリビューションとは「寄付」と訳します。「*LIBC*に対してプログラムやデータを無償で提供する」という意味でとらえてください。

**◆ バージョンアップ**

***Project LIBC Group***は，できる範囲内で*LIBC*のバグフィクス，機能拡張，追加などの**バージョンアップ**作業を随時行っていきたいと思っています。方法としては，全体の配布，差し替えファイルの提供，ソースコードのテキスト差分，バイナリ差分などがあります。これらのバージョンアップは，NIFTY-ServeのSHARP USER'Sフォーラム・ワークステーション館（FSHARP3）にて行う予定です。もし，*LIBC*に対する要望や問い合わせ，バグに関する情報があれば，次に示す著者のIDに電子メールを送ってください。

**NIFTY-Serve … PGA01555 (PGA01555@niftyserve.or.jp)**

ただし，これらのバージョンアップ作業は***Project LIBC Group***が独自に行うものであり，本書の発行元であるソフトバンクとは無関係です。また，*X68k Programming Series #1 Develop.*の著者グループと共同で出版する予定の追補版[8]を除けば，シリーズの一冊として，書籍でバージョンアップ版を提供することはありません。

8)追補版ではX68030へのより細かい対応と，バグフィクスを行う予定です。

**バージョンアップはパソコン通信以外では行いません**

## ❖ *LIBC*と*XC*

最初に*LIBC*と*XC*との関係と，そのなかでの*LIBC*の位置付けについてはっきりさせておきます。つまり，*LIBC*だけでできることと，*LIBC*ではできないことがあるのです。*LIBC*と*XC*のそれぞれの特長を考えて利用することが重要になります。

**◆ *LIBC*は*XC*に置き換わるか**

*LIBC*は*XC*に置き換わるものではありません。確かに，当初は*LIBC*を*XC*と並ぶものとして，つまり*XC*を完全に置き換えるつもりで作成していました。しかし結論からいえば，*LIBC*にも*XC*にもそれぞれに得手不得手があり，*LIBC*さえあれば後は何もいらないとは一概にはいえません。

ただ念のために記述しますが，*LIBC*は*XC*に対して上位コンパチブルになるように設計されています。ですから一部の互換性のない関数を除けば[9]，*XC*でできることは*LIBC*でもできます。

9) *LIBC*は，*X-BASIC*については一切サポートしていません。

それでは両者の得手不得手を1つずつ見ていくことにしましょう。最初に断わっておきますが，これは筆者の主観的な意見です。ですから，使う人にとっては長所もまた短所になりうるし，その逆もありえるでしょう。どちらでもよいと思う人もいるかもしれません。しかし1つの判断基準にはなるのではないかと思います。

○　*LIBC*の長所

1. **ソースコードがすべて公開されている**

   *XC*ももちろんソースコードが公開されていますが，*LIBC*のソースコードは著作権を主張しないPDSとして公開されています。また，ソースコードの95%はC言語で書かれており，コメントも十分に書き込まれているために理解がしやすくメンテナンスも楽でしょう。

2. **ライブラリすべてがPDSである**

   *LIBC*は商品でもなければフリーウェアでもありません[10]。PDSとして著作権とは無縁のものです。したがって，誰でもどんな用途にでも自由に使うことができ，また自由に改変/コピーができます。

10) 詳細については前述した「PDSとは」(P.15)を参照してください。

3. **他の処理系との互換性が高い**

   *LIBC*は，当初からUNIXからの移植をしやすくするという目的がありましたから，UNIXにしかないような関数，インクルードファイル，システムコールを多く含んでいます。また同時に，MS–DOSとのソース互換性を高める意味でも，*MS–C 7.0*などの機能を採り入れています。

4. **さまざまな規格に対応している**

   *LIBC*は*ANSI C*の標準ライブラリの仕様を満足しているだけではなく，可能なかぎり，たとえば*POSIX.1*や*XPG3*，*4.3BSD*や*SYSV*の仕様(規格ではありませんが)など，さまざまな機能を取り込んで設計されています[11]。

11) 詳細については後述する「インクルードファイルの互換性」(P.21)を参照してください。

5. **ある程度有用なシグナル機構**

   *XC*ではほとんど意味をなしていなかったシグナル機構を，同一プロセス内にかぎってですが，有効に動作させるようにしてあります。これは同時に，UNIXからの移植性を高めることにも役立っています。

6. **パソコン通信で継続的なサポートがある**

   もしライブラリに不都合やバグがあれば，パソコン通信上でそれを随時修正していく予定です。また，機能の拡張や関数の追加も行われるでしょう。詳しくは前述した「バージョンアップ」(P.17)を参照してください。

### ○ *LIBC*の短所

**1. 実行ファイルが大きくなる**

*LIBC*は95%がC言語で書かれていること，機能が豊富なこともあって，ライブラリ自体の大きさが*XC*の1.5～2倍近くあります。特に小さいプログラムをコンパイルすると，できあがる実行ファイルの大きさの違いは顕著に現れます。たとえばList 1-1のような短いプログラムを例にすると，*XC*では10Kバイトちょっとの実行ファイルが，*LIBC*では20Kバイト近くになることもあります。機能とサイズでのトレードオフが必要でしょう。

**List 1-1 ● とても短いプログラム**

```
 1: /*
 2: ** hello.c:
 3: **   世界で1番有名なプログラム。画面上に "Hello world !!" と表示
 4: **   する
 5: */
 6: #include <stdio.h>
 7:
 8: int main (void)
 9: {
10:     printf ("Hello world !!\n");
11:     return 0;
12: }
```

**2. 実行速度がやや遅い**

*LIBC*は考えられるかぎり高速になるように設計してはいますが，*XC*と比較した場合，*XC*がオールアセンブラであり，機能的にも限定されている点を考慮すると「遅く」はありませんが,「やや遅い」とはいえるでしょう。使う関数にもよるので，一概にどれだけ遅いかは判定できません。場合によっては速くなる場合もあります。ですから，実際に使用してみることをお勧めします。

**3. 商品ではない**

*LIBC*が商品ではなく，PDSとして公開されていることは長所でもあり，逆に短所でもあります。*LIBC*は商品ではないために，悪くいえば「信用できない」し，サポートも「期待できない」ことになります。先ほどサポートは「する予定」と書いたものの,「期待できない」のもまた事実です。もっとも**X68000**ユーザならば，自分で何とでもしてしまう方が多いかもしれませんね。

**4. *X-BASIC*をサポートしていない**

*LIBC*は*XC*のように*X-BASIC*をサポートしていません。“**ベーシック→C コンバータ**”もありませんし，ベーシックライブラリ（“`baslib.a`”）やそのためのインクルードファイルも備えていません。ですから，スプライトやPCM，FM音源関数を使ってのプログラミングはできません。ただし，IOCSコールライブラリはありますから，どうしても必要なら

ば直接IOCSコールを使ってプログラミングしたり，I/Oを操作することは可能です。

**5. 数値演算コプロセッサがない場合はFLOATパッケージに依存する**

*LIBC*は数値演算コプロセッサが装備されていると，自動的にそれを使って数学関数の演算を行います。しかしコプロセッサがない場合は，必ずFLOATパッケージを必要とします。*XC*のように，ライブラリ内部に計算ルーチンをもたせることはできません。

**6. *XC*上で開発されたライブラリと互換性がない**

*LIBC*はソースレベルの互換性だけを考慮し，オブジェクトレベル，つまり“.a”ファイルや“.l”ファイルレベルの互換性は無視しました[12]。ですから，これまで*XC*の標準ライブラリとリンクすることを想定して作られたライブラリは，そのままではリンクできません。一度再コンパイルする必要があります。また，逆に*LIBC*のライブラリを*XC*にもっていくこともできません。なぜならば，内部構造が異なっているからです。

12) これは*LIBC*を作成するうえで最低限必要な選択でした。

### ◆ *LIBC*と*XC*の選択

以上で，*LIBC*にはさまざまな長所と短所のあることが理解できたかと思います。ですから，この*LIBC*の長所および短所を理解したうえで，*XC*なり*LIBC*なりどちらでも好きなほうを使えばよいのです。

たとえば，あなたがFM音源やスプライトを用いたゲームを作ろうと思っているならば*XC*を使うべきです。*XC*にはスプライトやFM音源，PCMを比較的簡単に扱える関数がたくさんあります。それらを使えば，*LIBC*を使うよりもより簡単にプログラムを作成できるでしょう。それに，おそらく*XC*のほうが速いと思います。

しかし，あなたが今UNIXやMS–DOS上のプログラムを移植しようとしたり，*ANSI C*に準拠した正しいプログラミングを学ぼうというのであれば，*LIBC*を選択するほうがよいかも知れません。*LIBC*は*ANSI C*の標準ライブラリに忠実ですし，もともとUNIXからの移植を行うために作られています。速度や実行ファイルのサイズに多少の犠牲は出ますが，それはやむを得ないと割り切ってください。

# 1.3……………インクルードファイルの使い方

## ❖インクルードファイルの互換性

*LIBC*は次の資料を基に開発されました。本書では，文中でそれぞれの資料を引用するために ***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***，***4.3BSD***，***MS–C 7.0*** という表記を用いています。

○ ***ANSI C***

**米国規格協会** (ANSI – *American National Standards Institute*) の X3J11 委員会で 1989 年 12 月 14 日に承認，制定された C 言語の標準化案。ANSI X3.159–1989, *Programming Language C.*

○ ***POSIX.1***

**米国電気電子学会** (IEEE – *the Institute of Electrical and Electronics Engineers, Inc.*) が制定したオペレーティングシステムに関する標準化案。POSIX (IEEE Std 1003.1–1988, *Portable Operating System Interface for Computer Environments.)* および IEEE Std 1003.1–1990, *Information Technology–Portable Operating System Interface (POSIX)–Part1: System Application Program Interface (API)* [C Language].

○ ***XPG3***

X/Open Company Limited. が作成したコンピュータ環境に関する標準化案。*X/Open Portability Guide Issue 3, Volume 2–1989 XSI System Interface* and Headers.

○ ***AES/OS***

OSF (Open Software Foundation) が作成したオペレーティングシステムに関する標準化案。Application Environment Specification (AES) Operating System Programming Interfaces Volume, *Revision A.*

○ ***SYSV***

**米国電信電話会社** (AT&T – American Telephone and Telegraph Company) が作成した UNIX System V に関する次の資料。*SVID Issue 1 – System V Interface Definition, Issue 1*, Spring 1985 および *SVID Issue 2 – System V Interface Definition Issue 2*, 1986 および *SVID89 – System V Interface Definition, Issue 3*, Industry Review Draft, December 21, 1988.

- ○ ***4.3BSD***
  米国カリフォルニア大学バークレー校で作成された **UNIX** のバージョン 4.3BSD に関する次の資料。*UNIX Programmer's Reference Manual*, 4.3 Berkeley Software Distribution (BSD), April, 1986.
- ○ ***MS–C 7.0***
  米国マイクロソフト社 (*Microsoft Corporation*) が販売している C/C++コンパイラ，Microsoft C/C++ のマニュアル。Microsoft C/C++ Run-Time Library Reference Covers version 7.

## ❖ インクルードファイル一覧

***LIBC***のインクルードファイルは***ANSI C***，***XC***，***MS–C 7.0***，***POSIX.1***，***XPG3***，***AES/OS***などの規格で定義されているインクルードファイルに加え，***4.3BSD***や***SYSV***がもっているインクルードファイルにも対応しています。本セクションでは，これらの数多くのインクルードファイルを 9 つのカテゴリに分類して，解説します。

### ◆ *ANSI C*インクルードファイル

***LIBC***は***ANSI C***で規定されているインクルードファイルをすべて備えており，それぞれ次のようなことがらを宣言あるいは定義しています。これらのファイルは，ほとんどすべての処理系 (もちろん ***XC***にも存在していますから) で使用するのに制限はありません。ただし，処理系によっては ***ANSI C***への対応がやや異なっている場合があります。

- `assert.h`　プログラム診断を行うマクロの定義
- `ctype.h`　文字の分類を行う関数とマクロの宣言
- `errno.h`　変数 `errno` とその値を表現するマクロの宣言
- `float.h`　`float` 型，`double` 型，`long double` 型それぞれの浮動小数表現の表現範囲の定義
- `limits.h`　`char` 型，`short` 型，`long` 型それぞれの表現範囲を始めとして，いろいろな値の制限値の定義
- `locale.h`　ロケールについての定義。ただし，***LIBC***では C ロケールしか対応していないため，実質的には機能しない
- `math.h`　数学関数の宣言
- `setjmp.h`　大域ジャンプの定義
- `signal.h`　シグナル機構の定義
- `stdarg.h`　可変長引数を取り扱うマクロの定義
- `stddef.h`　***ANSI C***で使用される標準データ型とマクロの定義
- `stdio.h`　高水準ファイル入出力である `stdio` ライブラリの定義

- `stdlib.h`　　ほかの分類に属さない，その他いろいろの関数の宣言
- `string.h`　　文字列を扱う関数の宣言
- `time.h`　　時間を扱う関数の宣言

### ◆ *XC*インクルードファイル

*LIBC*は*XC*とのソースレベルの互換性を考慮して設計しており，*XC*が備えているインクルードファイルのうち，*LIBC*で代用可能なものについては互換ファイルがあります。これらの互換ファイルは関数名の置き換えを行っているものもあれば，同じ働きをする別のファイルを読み込むだけのものもあります。最低限の互換性は保証していますが，完全に同じとはかぎりません。あらかじめ了解しておいてください。

- `conio.h`　　コンソール直接入出力を扱う関数の宣言
- `doslib.h`　　DOSコールを扱う関数を宣言しているが，*LIBC*では代わりに`<sys/dos.h>`を用いるのが普通
- `io.h`　　低水準入出力を扱う関数を宣言しているが，*LIBC*では代わりに`<unistd.h>`を用いるのが普通
- `iocslib.h`　　IOCSコールを扱う関数を宣言しているが，*LIBC*では代わりに`<sys/iocs.h>`を用いるのが普通
- `jctype.h`　　シフトJISコードの日本語文字を分類する関数とマクロを定義しているが，*LIBC*では代わりに`<mbctype.h>`を用いるのが普通
- `jstring.h`　　シフトJISコードの日本語文字列を扱う関数を宣言しているが，*LIBC*では代わりに`<mbstring.h>`を用いるのが普通
- `process.h`　　子プロセスを呼び出す関数を宣言しているが，*LIBC*では代わりに`<unistd.h>`を用いるのが普通
- `sys/timeb.h`　　現在時刻を得る関数の宣言

### ◆ *MS–C 7.0*インクルードファイル

*LIBC*は*MS–C 7.0*で作成されたソースファイルを，なるべく少ない変更で動作させることができるように設計されており，*MS–C 7.0*が備えているインクルードファイルのうち，*LIBC*で代用可能なものについては互換ファイルがあります。しかし，この*MS–C 7.0*との互換性はおもに関数レベルで行っており，これまで紹介してきたインクルードファイル中にすでに含まれていますから，インクルードファイル自体は多くありません。

- `mbctype.h`　　シフトJISコード体系の日本語文字を分類する関数とマクロの宣言。*MS–C 6.0*以前のバージョンとの互換を保つには`<jctype.h>`を用いる
- `mbstring.h`　　シフトJISコード体系の日本語文字列を扱う関数の宣言。*MS–C 6.0*以前のバージョンとの互換を保つには`<jstring.h>`を用いる

- sys/locking.h　　ファイルロックを行う関数の宣言

**◆ *POSIX.1* インクルードファイル**

*LIBC* は *POSIX.1* をベースにして作成しており，*POSIX.1* で規定されているインクルードファイルをほとんど備えています。ですから，*POSIX.1* に対応して作成されたソースファイルとの互換性はある程度保たれています。ただし，インクルードファイル中には将来実現されることを見込んで作成したものもあり，実際には何ら機能しないものもいくつか存在します。また，これらのファイル中では，規格チェックのための_POSIX_SOURCE マクロをチェックしていません。

- dirent.h　　ディレクトリエントリを検索する関数の宣言
- fcntl.h　　ファイルのオープンやファイルハンドルの操作を行う関数の宣言
- grp.h　　グループファイルを扱う関数の宣言
- pwd.h　　パスワードファイルを扱う関数の宣言
- sys/stat.h　　ファイルの情報を取得する関数の宣言
- sys/times.h　　現在は機能しない
- sys/types.h　　*POSIX.1* で使用するデータ型とマクロの定義
- sys/utsname.h　　システムに関する情報を取得する関数の宣言
- sys/wait.h　　現在は機能しない
- termios.h　　現在は機能しない
- unistd.h　　低水準入出力など UNIX システムコールに関する宣言

**◆ *XPG3* インクルードファイル**

*LIBC* は *XPG3* についてもある程度対応して作成しており，*XPG3* で規定されているインクルードファイルを備えています。ですから，*XPG3* に対応して作成されたソースファイルとの互換性はある程度保たれています。ただし，これらのインクルードファイルは将来実現されることを見込んで作成したものであり，実際には何ら機能しないものがほとんどです。また，これらのファイル中では，規格チェックのための_XOPEN_SOURCE マクロをチェックしていません。

- ftw.h　　現在機能しない
- langinfo.h　　現在機能しない
- nl_types.h　　現在機能しない
- regexp.h　　現在機能しない
- search.h　　現在機能しない
- ulimit.h　　現在機能しない
- varargs.h　　可変長引数を取り扱う古いスタイルのマクロの定義

### ◆ *AES/OS*と *SVR4*

*LIBC*は *AES/OS*や *SVR4*についてもある程度対応して作成しており，*AES/OS*で規定されているインクルードファイルを備えています。ですから，*AES/OS*や*SVR4*に対応して作成されたソースファイルとの互換性は，多少なりとも保たれています。ただし，これらのインクルードファイルは将来実現されることを見込んで作成したものであり，実際には何ら機能しないものがほとんどです。また，これらのファイル中では，規格チェックのための_AES_SOURCE マクロをチェックしていません。

- poll.h　　現在機能しない
- sys/mman.h　　現在機能しない
- sys/timers.h　　現在機能しない
- wctype.h　　幅広文字を分類する関数とマクロの定義
- widec.h　　幅広文字を stdio ライブラリで扱うための関数の宣言
- wstring.h　　幅広文字列を扱う関数の宣言

### ◆ その他の互換ファイル

*LIBC*ではこれまでに紹介したインクルードファイル以外にも *SYSV*や *4.3BSD*のソースコードとの互換性を考え，いくつかの互換ファイルを提供しています。ただし，これらの互換ファイルは関数レベルの互換性というよりは，インクルードファイルの名前やディレクトリ中の位置が異なるのをカバーするためのものです。

- alloca.h　　alloca 関数の宣言
- malloc.h　　<stdlib.h>の別名
- memory.h　　<string.h>の別名
- strings.h　　<string.h>の別名
- sys/dir.h　　<dirent.h>の別名
- sys/exec.h　　<a_out.h>の別名
- sys/fcntl.h　　<fcntl.h>の別名
- sys/file.h　　<fcntl.h>の別名
- sys/param.h　　<limits.h>の別名であるとともに，このファイル特有のマクロをいくつか定義している
- sys/signal.h　　<signal.h>の別名
- sys/time.h　　現在は機能しない
- sys/resource.h　　システムリソースを操作する関数の宣言

### ◆ *LIBC*インクルードファイル

最後に *LIBC*特有のインクルードファイルを説明しておきます。これらのファイルは *LIBC*に特有のものなので，もし他の処理系，たとえば *XC*，*MS–C 7.0*，UNIX などへの移植を考えてプログラミングするならば，決して使用しないでください。

- a_out.h　X型の実行ファイルと *LIBC* が生成する "core" ファイルの構造の定義
- interrupt.h　GCC で割り込みを記述するために必要なマクロの定義
- wchar.h　幅広文字を扱うための宣言
- sys/dos.h　DOS コールライブラリの宣言
- sys/iocs.h　IOCS コールライブラリの宣言
- sys/project.h　*LIBC* のバージョンを求めるための関数の宣言
- sys/scsi.h　SCSI コールライブラリの宣言

### ◆ *LIBC* の内部ファイル

次に示したファイルは，*LIBC* を作成するためのライブラリ内部のインクルードファイルですから，普通は使用しないでください。これらのファイルには移植性がなく，しかも内容が将来変更される可能性があります。ライブラリの内部構造をよく理解している人や，ライブラリを作成する人のみ使用するようにしてください。

- cdecl.h　すべてのインクルードファイルに読み込まれるファイル
- sys/dos_i.h　DOS コールをインライン展開するための宣言
- sys/dos_p.h　DOS コールのプロトタイプの宣言
- sys/iocs_i.h　IOCS コールをインライン展開するための宣言
- sys/iocs_p.h　IOCS コールのプロトタイプの宣言
- sys/xdoseml.h　DOS エミュレーション関数のためのファイル
- sys/xglob.h　ライブラリ用のグローバル関数のためのファイル
- sys/xgrp.h　グループファイル関数のためのファイル
- sys/xmath.h　数学関数のためのファイル
- sys/xmbstring.h　マルチバイト文字列関数のためのファイル
- sys/xprof.h　プロファイラのためのファイル
- sys/xpwd.h　パスワードファイル関数のためのファイル
- sys/xresource.h　リソース関数のためのファイル
- sys/xsignal.h　シグナル機構のためのファイル
- sys/xstart.h　スタートアップ関数のためのファイル
- sys/xstat.h　stat 関数のためのファイル
- sys/xstdio.h　stdio ライブラリのためのファイル
- sys/xstdlib.h　標準ライブラリのためのファイル
- sys/xtime.h　時間関数のためのファイル
- sys/xunistd.h　UNIX システムコールエミュレーションのためのファイル

## ❖ インクルードファイルの使い方

*LIBC*のインクルードファイルは，GCC でも *XC*でも使用することができます。本セクションではそれぞれのコンパイラで，*LIBC*を利用する手順について説明します。

### ◆ GCC で使う

基本的には，この *LIBC*は GCC で使用してください。*XC*でも使用することはできますが，*XC*では十分なテストを行っていません。さらに *XC*では，*LIBC*の機能を十分に引き出すことは無理でしょう。

GCC で *LIBC*を使うためには，*LIBC*の include ディレクトリの位置を環境変数 include に設定する必要があります。たとえば，*LIBC*をインストールするときに，インストール先ディレクトリとして "A:\usr\lang" を指定したならば，環境変数 include には "A:\usr\lang\include" と設定してください。もちろんパス名の区切りとして，"\" のほかに "/" を使うこともできます。その場合は，"A:/usr/lang/include" となります。

もしあなたがコマンドシェルとして COMMAND.X を使っているならば，自分の "AUTOEXEC.BAT" に次の 1 行を加えてください。

```
set include=A:\usr\lang\include
```

### ◆ *XC*で使う

*XC*で *LIBC*を使う方法は，GCC で *LIBC*を使う場合と基本的には同じです。しかし環境変数 include に設定するパス名の区切り記号は，必ず "\" を使用してください。"/" も使えるかも知れませんが，"\" のほうが無難です。そして同じように，"AUTOEXEC.BAT" に次の 1 行を加えてください。

```
set include=A:\usr\lang\include
```

重ねて説明しますが，*XC*では *LIBC*のすべての機能を使用することができません。できるだけ，GCC と組み合わせてお使いください。

**LIBCは GCC で使用してください**

## ❖インクルードファイルについて

*LIBC*のインクルードファイルは表面的な仕様以外にも，たくさんの仕様が含まれています。この機能をより有効に利用するために，次の説明と注意点をよく読んでおいてください。

### ◆ 関数のプロトタイプ

*LIBC*は*ANSI C*ベースで開発されることを想定しており，古い***K&R***時代のC言語にしか対応していないコンパイラ[1)]での使用は想定していません。すべての関数は厳格に**プロトタイプ**宣言してあり，引数の型まちがいを適切に判断できるようになっています。またポインタに対しては，必要に応じてconst型として宣言してありますから，不用意な領域書き換えも防げます。

1)X68000上ではこのようなコンパイラはありません。

特に**GCC**といっしょに使用するときには，**GCC**の拡張機能である**関数属性**も正しく評価されるように工夫してあります。**副作用のない関数**には__const型の関数属性を，また呼出元に**戻ってこない関数**には__volatile型の関数属性を宣言しています。

### ◆ C++言語で使用するには

*LIBC*は，C++言語で使用されることを最初から考慮してあります。*LIBC*のインクルードファイルは__cpluscplusマクロが定義されているかどうかによって，C言語かそれともC++言語かどうかを判断し，C++言語で使用される場合にはライブラリ関数のプロトタイプをすべて，extern "C" {}によって囲むように変化します。

このように，*LIBC*はC++言語で使用されることを想定してはいますが，C++特有の関数[2)]については宣言していません。また，<stream.h>などのような**クラスライブラリ**などもありません。それらのライブラリは別途用意する必要があります。たとえば，FSF (*Free Software Foundation*) のGNU**プロジェクト**で作成された“libg++”クラスライブラリやNIH (*National Institute of Health*) の“nihcl”クラスライブラリ[3)]などを移植して，利用されるとよいでしょう。

2)たとえばnewやdelete文を処理する内部関数など。

3)これらはいずれもフリーウェアとして公開されています。

### ◆ 関数とマクロの切り替え

*LIBC*では，提供している関数のうちマクロとして定義できるものについては関数版とマクロ版の両方を提供しており，用途に応じて使い分けられるようになっています。デフォルトではほぼすべてのものについてマクロ版が使われるように設定されていますが，もし関数版を用いたい場合は，必要に応じて次の**選択マクロ**を#define で定義してください。たとえばある関数のアドレスを得る必要がある場合には，その関数は実体をもつ“**関数版**”でなければなりません。ただし，これらの選択マクロは，必ずインクルードファイルを読み込む前に定義してください。読み込んだ後に定義してもまったく意味をなしません。

また，これと同様にインライン関数として定義したほうがよいものについても，関数版とインライン版の両方を提供しています。必要に応じて使い分けるとよいでしょう。

- `__NO_LOCALE_INLINE__`　**ロケール**関数をマクロ展開しないようにする。デフォルトではマクロ展開
- `__NO_CTYPE_INLINE__`　文字を分類する関数をマクロ展開しないようにする。デフォルトではマクロ展開。これを定義すると，<`ctype.h`>で定義されている文字分類のほかに<`mbctype.h`>，<`jctype.h`>，<`wctype.h`>も同様に影響される
- `__NO_SIGNAL_INLINE__`　シグナル関数をマクロ展開しないようにする。デフォルトではマクロ展開
- `__NO_STDIO_INLINE__`　`stdio` **ライブラリ**をマクロ展開しないようにする。デフォルトではマクロ展開
- `__NO_STDLIB_INLINE__`　`stdlib` 標準ライブラリをマクロ展開しないようにする。デフォルトではマクロ展開
- `__DOS_INLINE__`　DOS コールをインライン展開する。デフォルトではインライン展開はしない。ただし，インライン展開には非常に処理時間がかかる
- `__IOCS_INLINE__`　IOCS コールをインライン展開する。デフォルトではインライン展開はしない。ただし，インライン展開には非常に処理時間がかかる

たとえば次の List 1-2 を見てください。このプログラム中の文字分類を行う関数を，マクロ展開しないようにコンパイルするには，6 行目のような記述が必要です。

**List 1-2 ● 文字分類を行うプログラム**

```
 1:  /*
 2:  ** isupper.c:
 3:  **    キーボードから文字列を入力してもらい，その文字列の
 4:  **    先頭が英大文字かどうかを調べる
 5:  */
 6:  #define __NO_CTYPE_INLINE__ /* <-- これが必要 */
 7:
 8:  #include <stdio.h>
 9:  #include <ctype.h>
10:
11:  int main (void)
12:  {
13:      char buffer[256];
14:
15:      while (gets (buffer) != NULL) {
16:          if (isupper (*buffer))
17:              printf ("YES!\n");
18:          else
19:              printf ("NO..\n");
20:      }
```

```
21:
22:         return 0;
23:  }
```

上記とは別の方法として，直接コンパイラに選択マクロを与えることでも同じ結果が得られます。次の画面の表示例では，**GCC** を使い，最適化ありでコンパイルしています。

```
A:\> gcc -O -D__NO_CTYPE_INLINE__ isupper.c
```

# 1.4 ……………… ライブラリファイルの使い方

## ❖ ライブラリファイル一覧

*LIBC*は次の 11 のライブラリファイルを標準で提供しています。すべてのライブラリファイルは *XC* ver.1 で使用されていた “.a” 形式で提供されていますが，もしあなたが *XC*をもっているならば，LIB.X を用いて *XC* ver.2 相当の “.l” 形式にして使うと，より高速にリンクすることができます。

- libc.a　C 言語の標準ライブラリおよびそれに類するもの
- libcplus.a　C++言語用の追加ライブラリ
- libdos.a　DOS コールライブラリ
- libiocs.a　IOCS コールライブラリ
- libmb.a　日本語文字列処理ライブラリ
- libprof.a　GCC 用のプロファイルライブラリ
- libscsi.a　SCSI コールライブラリ
- libsignal.a　シグナルライブラリ
- libsuper.a　スーパーバイザモード実行用ライブラリ
- libtz.a　フルデコードタイムゾーンライブラリ
- libw.a　幅広文字ライブラリ

では，それぞれのライブラリについて，もう少し詳しく説明します。

### ◆ libc.a

C 言語の標準ライブラリとそれに類するものが含まれています。普通にプログラミングするだけならば，このライブラリだけで十分です。このライブラリは何も指定しなくても，**コンパイラドライバ**[1]によって自動的にリンクされますから，**GCC** なり *XC*のマニュアル通り，普通にコンパイルすれば大丈夫です。

1)GCC なら gcc.x，*XC* なら cc.x のことを指します。

たとえば “libc.c” というプログラムがあるとします。このプログラムを **GCC** を使い，最適化ありでコンパイルするには次のようにタイプします。

```
A:\> gcc -O libc.c
```

◆ libcplus.a

C++言語をサポートするライブラリです。C++言語でプログラムする場合には，必ずこのライブラリをリンクしてください。このライブラリをリンクしないと，*LIBC*は起動時と終了時に**グローバルコンストラクタ**や**グローバルデストラクタ**の呼び出しを行いません。このライブラリは g++[2]を使うかぎり，何も指定しなくても自動的にリンクされます。

2)GNU プロジェクトが提供している C++コンパイラ。

実際にグローバルコンストラクタとグローバルデストラクタを使ったプログラムを見てみましょう。次の List 1-3 はグローバルコンストラクタ，クラスメソッド，デストラクタを使って文字列を表示するプログラムです。

**List 1-3 ● C++のサンプルプログラム**

```
 1:  /*
 2:  ** cplus.c:
 3:  **   C++ で Global Constructor/Destructor と Class Method を
 4:  **   使って文字列を表示する。正しく動作すれば "Beauty and the
 5:  **   Beast" と表示される
 6:  */
 7:  #include <stdio.h>
 8:
 9:  class Sample {
10:  public:
11:      Sample () { printf ("Beauty"); }         // Ctors
12:     ~Sample () { printf ("the Beast\n"); }   // Dtors
13:      void and () { printf (" and "); }        // Method
14:  };
15:
16:  Sample belle; // Class Sample のグローバルインスタンス
17:
18:  int main ()
19:  {
20:      belle.and ();
21:      return 0;
22:  }
```

このプログラムをコンパイルするには，GNU C++コンパイラである **g++**を用います。基本的な使い方は **GCC** と同じです。List 1-3 を最適化ありでコンパイルするには次のようにタイプしてください。

```
A:\> g++ -O cplus.c
A:\> cplus.x
Beauty and the Beast
A:\>
```

### ◆ libdos.a

DOSコールライブラリが含まれています。DOSコールを使ったプログラムを作成する場合は，このライブラリをリンクしてください。このライブラリは**GCC**, *XC*いずれの場合も自動的にはリンクされませんから，自分で明示的に指定する必要があります。**GCC**ならば，“-ldos”というオプションを指定してください。

たとえば，DOSコールを使用する“dos.c”というプログラムがあるとします。これを**GCC**を使い，最適化ありでコンパイルするには次のようにタイプします。

```
A:\> gcc -O dos.c -ldos
```

### ◆ libiocs.a

IOCSコールライブラリが含まれています。IOCSコールを使ったプログラムを作成する場合は，このライブラリをリンクしてください。ただし*XC*とは異なり，SCSI装置を扱うためのSCSIコールが別のライブラリになっています。SCSIコールライブラリについては，後述するlibscsi.aを参照してください。また，このライブラリは**GCC**, *XC*いずれの場合も自動的にはリンクされませんから，自分で明示的に指定する必要があります。**GCC**ならば，“-liocs”というオプションを指定してください。

たとえば，IOCSコールを使用する“iocs.c”というプログラムがあるとします。これを**GCC**を使い，最適化ありでコンパイルするには次のようにタイプします。

```
A:\> gcc -O iocs.c -liocs
```

### ◆ libmb.a

*MS–C 7.0*互換の日本語文字処理ライブラリ，つまりシフトJIS**マルチバイトコード**の日本語文字列を扱う関数が含まれています。“mb”というのはマルチバイトを意味しています。これらの関数を使ったプログラムを作成する場合は，このライブラリをリンクしてください。このライブラリは**GCC**, *XC*いずれの場合も自動的にはリンクされませんから，自分で明示的に指定する必要があります。**GCC**ならば，“-lmb”というオプションを指定してください。

たとえば，マルチバイト関数を使用する“multi.c”というプログラムがあるとします。これを**GCC**を使い，最適化ありでコンパイルするには次のようにタイプします。

```
A:\> gcc -O multi.c -lmb
```

◆ libprof.a

GCCの“-p”オプションや“-a”オプションで利用するプロファイラが含まれています。このライブラリは，GCCでこれらのオプションを指定したときに自動的にリンクされます。ただし，このライブラリはGCCの機能を利用するためのものなので，*XC*では使用できません。

プロファイラを用いると，“-p”オプションの場合ではどの関数が何回呼び出され，実際にどれだけの時間を消費したかを，また“-a”オプションの場合ではどの論理ブロックを何回通過したかを統計として計算し，プログラム終了時に標準出力に出力します。

それでは，次のList 1-4というプログラムを使ってプロファイラの実際の使用例を見てみましょう。

**List 1-4 ● プロファイルライブラリの使用例**

```
 1:  /*
 2:  ** profile.c:
 3:  **    1から5までの数字を表示する
 4:  */
 5:  #include <stdio.h>
 6:
 7:  void hello (int i)
 8:  {
 9:      printf ("count = %d\n", i);
10:  }
11:
12:  int main (void)
13:  {
14:      int i;
15:
16:      for (i = 0; i < 5; i++)
17:          hello (i);
18:      return 0;
19:  }
```

上記のプログラムをGCCの“-p”オプションをつけてコンパイルする手順と，できあがった実行ファイルの実行結果を下図に示します。

```
A:\> gcc -O -p profile.c
A:\> profile
count = 0
count = 1
count = 2
count = 3
count = 4
count = 5
Function          main :  0.000 sec(s) :      1 time(s)
Function         hello :  0.000 sec(s) :      5 time(s)
A:\>
```

この表示例では，main関数が1回，hello関数が5回実行され，それぞれの関数で消費した時間が0.000秒(測定限界以下)だったことを表しています。

ただし，プロファイラの測定は1/100秒単位であり，ほとんど瞬時に通過してしまうような短い関数では，かなり測定誤差が出てしまいます。このことは，あらかじめ了解しておいてください。

◆ libscsi.a

SCSI装置を操作するSCSIコールライブラリが含まれています。SCSIコールを使ったプログラムを作成する場合は，このライブラリをリンクしてください。*XC*では，このSCSIコールライブラリは**DOS**コールライブラリに含まれますが，*LIBC*では独立したライブラリです。このライブラリは**GCC**，*XC*いずれの場合も自動的にはリンクされませんから，自分で明示的に指定する必要があります。**GCC**ならば，“-lscsi”というオプションを指定してください。

たとえば，SCSIコールを使用する“scsi.c”というプログラムがあるとします。これを**GCC**を使い，最適化ありでコンパイルするには次のようにタイプします。

```
A:\> gcc -O scsi.c -lscsi
```

◆ libsignal.a

*LIBC*は同一プロセス内にかぎってですが，***POSIX.1***で規定されたシグナル機構に似たシグナルインタフェイスを提供しています。しかし，この機能は非常に微妙な位置付けにあります。

*LIBC*ではこのシグナルインタフェイスをエミュレートするために割り込みを使用したり，システムの割り込みベクタを書き換えるなど，やや危険な処理をしています。また，*LIBC*のシグナルインタフェイスはサイズ的にも大きくなっています。そこで標準では，この機能は組み込まないような設定になっています。もしシグナルインタフェイスを利用したいのならば，自分で明示的に指定する必要があります。**GCC**ならば，“-lsignal”というオプションを指定してください。

たとえば，シグナルインタフェイスを使った“signal.c”というプログラムがあるとします。これを**GCC**を使い，最適化ありでコンパイルするには次のようにタイプします。

```
A:\> gcc -O signal.c -lsignal
```

◆ libsuper.a

スーパーバイザモードでの実行をサポートするライブラリが含まれています。普通，*LIBC*を使用して作成されたプログラムは**ユーザモード**で実行されますが，こ

のライブラリをリンクすると，最初から最後までスーパーバイザモードで実行されるようになります。

これは特に **X68000** で，数値演算プロセッサボードをもっている場合に有効です。***LIBC*** は起動時に数値演算プロセッサボードの有無を調べ，もしボードがあれば MC68881 を I/O 経由で直接駆動しますが，このときスーパーバイザモードとユーザモードの切り替えが発生します。このライブラリを用いて最初からスーパーバイザモードで実行すれば，切り替える必要がなくなるので，高速に実行できるようになります[3]。ただし **X68030** では I/O を経由せず，直接 MC68882 に浮動小数計算させることができるため，このライブラリは **X68030** では意味がありません。

3)ただし，MC68000 について十分な知識がないなら，このライブラリを使ってスーパーバイザモードで実行することは避けてください。

このライブラリは **GCC**，***XC*** いずれの場合も自動的にはリンクされませんから，自分で明示的に指定する必要があります。**GCC** ならば，“-lsuper” というオプションを指定してください。

たとえば，“fastmath.c” というプログラムがあるとします。これを **GCC** を使い，最適化ありでコンパイルするには次のようにタイプします。

```
A:\> gcc -O fastmath.c -lsuper
```

◆ libtz.a

***LIBC*** は，***POSIX.1*** で規定されているタイムゾーンの処理をサポートしています。しかしごく普通にプログラミングする場合は，タイムゾーンについてそれほど意識することはありませんし，日本以外の時間帯を気にして使用することもあまりありません。そこで ***LIBC*** では，環境変数 TZ を用いたタイムゾーンの処理を2通り用意してあります。

1つは ***POSIX.1*** の機能をすべてサポートしたもの，そしてもう1つは日本で使用することを前提とした固定処理タイプのものです。後者は固定的な処理とはいえ，日本国内で使用するぶんには問題ありませんし，前者のサイズが非常に大きいこともあり，***LIBC*** では，標準では後者の固定タイプが使用されています。もし，***POSIX.1*** タイプの完全なタイムゾーン処理を利用したいのならば，自分で明示的に指定する必要があります。**GCC** ならば，“-ltz” というオプションを指定してください。

たとえば，タイムゾーン処理を使った “time.c” というプログラムがあるとします。これを **GCC** を使い，最適化ありでコンパイルするには次のようにタイプします。

```
A:\> gcc -O time.c -ltz
```

◆ libw.a

幅広文字を扱うライブラリのうち，*SVR4*や*MSE*で定義されているものが含まれています。幅広文字を使ったプログラムを作成する場合は，このライブラリをリンクしてください。このライブラリはGCC，*XC*いずれの場合も自動的にはリンクされませんから，自分で明示的に指定する必要があります。GCCならば，“-lw”というオプションを指定してください。

たとえば，幅広文字を扱う関数を使った“wchar.c”というプログラムがあるとします。これをGCCを使い，最適化ありでコンパイルするには次のようにタイプします。

```
A:\> gcc -O wchar.c -lw
```

## ❖ライブラリファイルの使い方

*LIBC*のライブラリファイルは，GCCでも*XC*でも使用することができます。本セクションではそれぞれのコンパイラで，*LIBC*を利用する手順について説明します。

◆ GCCで使う

GCCで*LIBC*を利用するには，いくつかの設定が必要です。まず，*LIBC*のlibディレクトリの位置を，環境変数libに設定しなければなりません。たとえば，*LIBC*をインストールするときに，インストールディレクトリとして“A:\usr\lang”を指定したならば，環境変数libには“A:\usr\lang\lib”を設定してください。なおパス名の区切りには，“\”のほかに“/”を使うこともできます。

次に環境変数GCC_LIBを設定します。これはGCCがリンクするときに使用するライブラリの形態を指定するものです。*LIBC*が標準で提供しているライブラリは，*XC* ver.1の“.a”形式のライブラリファイルなので“.a”を指定します。もしあなたが*XC*をもっているならば，LIB.Xを使って“.a”形式のライブラリファイルを，より高速な*XC* ver.2相当の“.l”形式のファイルに変換することができます[4)]。この場合は環境変数GCC_LIBには“.l”を指定します。

最後にライブラリのネーミング方法について指定します。C言語の標準ライブラリを例にすると，*XC*ではこのライブラリは“clib.a”というファイル名です。しかし*LIBC*では前後を入れ換えて“libc.a”としています。このネーミング方法はUNIXにしたがったものです。同様に*XC*のDOSコールライブラリは“doslib.a”ですが，*LIBC*では“libdos.a”です。GCCが正しいライブラリをリンクできるように，またGCCの“-l”オプションを使って必要なライブ

4)プログラムLIB.Xの使い方については，*XC*のマニュアルを参照してください。

5)この方法での利用を強く勧めます。

ラリを選択できるよう，どちらのネーミング方法を使うかを指定しなければなりません。もしあなたが *X68k Programming Series #1 Develop.* に付属している **GCC** を利用しており，***LIBC***が標準で指定したファイル名のままで利用したければ[5)]，環境変数 GCC_NO_XCLIB に “yes” を指定してください。しかし，もしあなたが古いバージョンの **GCC** しかもっていないか，または ***XC***と同じネーミング方法を用いたければ，環境変数 GCC_NO_XCLIB を定義しないようにして，***LIBC*** のライブラリファイルを ***XC***形式にリネームしてください。

以上のことをふまえ，自分の AUTOEXEC.BAT を書き換えます。もしあなたがコマンドシェルとして COMMAND.X を使っているならば，以下の行を加えてください。

```
set lib=A:\usr\lang\lib
set GCC_LIB=.a
set GCC_NO_XCLIB=yes
```

また，***LIBC***のライブラリファイルを ***XC***形式の名前に変えるには，インストールした ***LIBC***の bin ディレクトリにある “LIBNAME.BAT” を実行してください。次のような画面が表示され，ライブラリ名が ***XC***形式に変更されます。

```
A:\> CD A:\usr\lang\bin
A:\usr\lang\bin> LIBNAME.BAT

ライブラリの名称を XC 形式に変更します。

変更 libc.a --> clib.a
変更 libdos.a --> doslib.a
    :
    :
  中略
    :
    :
変更 libw.a --> wlib.a

変更は無事終了しました。

A:\usr\lang\bin>
```

### ◆ ***XC***で使う

まず前項の ***LIBC***を「GCC で使う」方法を読んでください。***XC***で使うには，このうちの環境変数 lib を設定することとライブラリファイルのファイル名を ***XC*** 形式に変更することが必要になります。このときは環境変数への設定において，パス名の区切り記号に必ず “\” を使用してください。

もしあなたが ***LIBC***のインストール先ディレクトリとして “A:\usr\lang” を指定し，コマンドシェルとして COMMAND.X を使っているならば，自分の AUTOEXEC.BAT に次の行を加えてください。

```
set lib=A:\usr\lang\lib
```

また，*LIBC*のライブラリファイルを*XC*形式の名前に変えるには，bin ディレクトリにあるLIBNAME.BATを実行してください。

```
A:\> CD A:\usr\lang\bin
A:\usr\lang\bin> LIBNAME.BAT

ライブラリの名称を XC 形式に変更します。

変更 libc.a --> clib.a
変更 libdos.a --> doslib.a
  :
  :
 中略
  :
  :
変更 libw.a --> wlib.a

変更は無事終了しました。

A:\usr\lang\bin>
A:\>
```

## ❖ ライブラリファイルについて

### ◆ *XC*との互換性

*LIBC*は*XC*の上位互換として作成されており，*XC*とはソースコードレベルで非常に互換性があります。ですから，これまで*XC*用に開発してきたソースコードを*LIBC*用に書き換えることは非常に容易です。場合によっては，何も書き換えなくてもよい場合もあるでしょう。

しかし，*LIBC*は*XC*のもつ機能をすべて備えているわけではありません。*LIBC*には*XC*のもっている機能のうち，次のものが欠けています。

- audio.h　PCM 音源を操作する関数
- graph.h　グラフィックスを描画する関数
- image.h　イメージユニットを操作する関数
- mouse.h　マウスを操作する関数
- music.h　FM 音源を操作する関数
- sprite.h　スプライトを操作する関数

● `stick.h`　　ジョイスティックを操作する関数

ただし，これらの高レベルの関数については少々手間はかかりますが，IOCSコールなどの低レベル関数によって代用することができます。

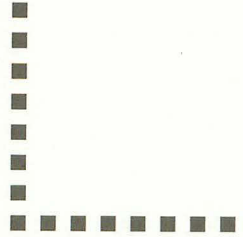
### ◆ *MS–C 7.0*との互換性

*LIBC*は*MS–C 7.0*特有の関数，たとえば日本語文字列の操作や`_fullpath`関数などのファイル操作関数を互換性のために提供しています。*MS–C 7.0*は，*MS–C 6.0*から関数名など細かい部分が変わっていますから，*MS–C 7.0*に関する参考書も役に立つと思います。

### ◆ *ANSI C*との互換性

*LIBC*は*ANSI C*で規定された標準ライブラリの仕様をほぼ満たしていますが，唯一，**ロケール**を十分にサポートしていません。*LIBC*でサポートしているロケールはCロケールだけであり，それも正しいロケール処理にのっとっているわけではなく，固定的にコーディングされています。ですからロケール関連の関数，たとえば`setlocale`関数，`strcoll`関数などを扱う場合には，このことを頭にいれておく必要があります。

また，*LIBC*では*ANSI C*で規定されている「*ANSI C*標準ではない関数はその関数名は“_”で始まるべきである」という条項を満たしていません。*MS–C 7.0*などでは厳密にこの条項を守っていますが，*LIBC*の場合，標準以外の関数が多すぎること，また従来のソースコードとの互換性などを考えたうえで，この条項を省くことにしました。

### ◆ *POSIX.1*との互換性

*LIBC*は*POSIX.1*で規定されたライブラリのうち，約60%程度の関数を提供しています。提供されていない残り40%の関数群は，たとえばプロセスのフォークであるとか，パイプの生成や端末操作といったOSに非常に依存するものなので，実現できていません。また，提供されている関数もすべての仕様を満足しているわけではなく，一部の機能は省略されています。これらについてはVol.2の「Programmer's Reference」を参照してください。

### ◆ サポートしていない関数

*LIBC*にはインクルードファイルに定義されているにもかかわらず，実体のない関数が数多くあります。これらの**未サポート関数**は，今の**Human68k**の使用では実現することができなかったり，実現できてもあまり使用されないだろうという判断で省かれているものです。**Human68k**の改良や**X68030**の性能によっては*LIBC*の将来のバージョンでサポートされる可能性もありますが，今の段階では決まっていません。

# Chapter 2
# *LIBC*プログラミング

本章では「*LIBC*プログラミング」と題して，*LIBC*ライブラリを用いたプログラミング技法をトピックスごとに解説します。もともとCの標準ライブラリには，誰もがほしいと思うような関数がほとんどそろっています。単にその存在を知らなかったり，使い方がわからなかったりして，ムダになっていることが意外と多いのではないでしょうか?

関数の「仕様」はVol.2で紹介するとして，ここでは具体的に，しかも普通はあまり詳しく触れられないようなトピックスについて，サンプルを交えて解説していきます。

# 2.1 ファイル

## ❖ ファイルを調べる

ファイルを作成したり，読み書きすることについてはよく書籍などで紹介されていますが，ファイルを調べる方法についてはあまり詳しく解説されていないようです。本セクションでは，ファイルとディレクトリの調べ方およびその扱い方について解説します。

### ◆ 指定したファイルを調べる

指定したファイルを調べる関数にはstat関数，fstat関数，lstat関数などがあり，<sys/stat.h>に定義されています。stat関数はパス名を指定してファイルを調べ，fstat関数は**ファイルハンドル**からファイルを調べます。lstat関数は少し特殊で，通常ファイルを指定するとstat関数と同一の動作をしますが，**シンボリックリンク**ファイルの場合，シンボリックリンク先のファイルではなく，シンボリックファイルそのものを調べます。これらの関数でファイルを調べると，Table 2-1のような**ファイルステータス**がわかります。このステータスは可能なかぎり，**UNIX**のstat関数の返すファイルステータスの内容に近付けましたが，**Human68k**のファイル構造では完全に**UNIX**と同一にはできませんでした。しかし*XC*のstat関数とは，上位互換[1]になっていますので，*XC*で求められるファイルステータスはすべて求めることができます。

1) *XC*とはソースレベルで互換性があります。

*XC*と比較して，特に拡張された仕様は以下のとおりです。

- **st_dev，st_rdevにより仮想ドライブ，仮想ディレクトリの判定が可能**
- **st_sizeはディレクトリサイズも求められる**
- **st_inoが疑似i-node番号を返す**
- **シンボリックリンク先がデバイスドライバの場合も調べられる**
- **ファイルモードがほぼUNIX互換になっている**
- **lstat関数がある**

簡単なstat関数の使用例をList 2-1に示します。

**Table 2-1 ● stat 構造体の内容**

| メンバ名 | 解 説 |
|---|---|
| dev_t st_dev | 物理ドライブ番号 |
| ino_t st_ino | 物理ドライブ×16777216+先頭セクタ番号 |
| mode_t st_mode | ファイルモード |
| nlink_t st_nlink | ファイルのリンク数を表すが，つねに1 |
| uid_t st_uid | ファイル所有者識別子を表すが，つねに0 (root) |
| gid_t st_gid | ファイルグループ識別子を表すが，つねに0 (root) |
| dev_t st_rdev | 論理ドライブ番号 |
| off_t st_size | ファイルサイズ，ディレクトリサイズ |
| time_t st_atime | 最終アクセス時間を表すが，つねに st_mtime と同じ |
| time_t st_mtime | 最終変更時間 (UTC) |
| time_t st_ctime | 最終状態変更時間を表すが，つねに st_mtime と同じ |

**List 2-1 ● stat 関数を用いたサンプルプログラム**

```
 1: /*
 2: ** sttest.c:
 3: **   "C:/COMMAND.X" を stat 関数で調べる
 4: */
 5: #include <stdio.h>
 6: #include <errno.h>
 7: #include <time.h>
 8: #include <sys/stat.h>
 9:
10: int main(void)
11: {
12:     /* 構造体の宣言 */
13:     struct stat st;
14:
15:     /* errno の初期化 */
16:     errno = 0;
17:
18:     /* stat 関数を呼び出す */
19:     if (stat ("C:/COMMAND.X", &st)) {
20:         perror ("stat");
21:         return;
22:     }
23:
24:     /* 戻り値の内容を表示する */
25:     printf ("  st_ino    : %08x\n", st.st_ino);
26:     printf ("  st_nlink  : %d\n", st.st_nlink);
27:     printf ("  st_uid    : %d\n", st.st_uid);
28:     printf ("  st_gid    : %d\n", st.st_gid);
29:     printf ("  st_rdev   : %d\n", st.st_rdev);
30:     printf ("  st_dev    : %d\n", st.st_dev);
31:     printf ("  st_mode   : %06o\n", st.st_mode);
32:     printf ("  st_size   : %d\n", st.st_size);
33:     printf ("  st_mtime  : %s\n", ctime (&st.st_mtime));
34:
35:     return 0;
36: }
```

このプログラムをコンパイルし実行させると，画面には次のように表示されます。

```
A:\> gcc -O sttest.c
A:\> sttest.x
  st_ino    : 02000050
  st_nlink  : 1
  st_uid    : 0
  st_gid    : 0
  st_rdev   : 2
  st_dev    : 2
  st_mode   : 100777
  st_size   : 28382
  st_mtime  : Thu Feb 25 12:00:00 1993
A:\>
A:\> DIR C:\COMMAND.X
 \HUMAN _Ver3          C:\
     1 ファイル      1006K Byte 使用中        215K Byte 使用可能
 ファイル使用量         28K Byte 使用
COMMAND         X        28382  93-02-25  12:00:00
A:\>
```

◆ ディレクトリを調べる

指定したディレクトリにどのようなファイルが存在するのか調べるには，**ディレクトリストリーム**を用います。ディレクトリストリームはstdioライブラリのファイルストリームと同じように，**ディレクトリエントリ**の構造をおおいかくし，簡単に操作できるようにするものです。

ディレクトリストリームを操作する関数にはopendir関数，readdir関数などがあり，<dirent.h>に定義されています(Table 2-2参照)。このディレクトリストリームのDIR構造体の構造はTable 2-3のように定義されていますが，通常，ユーザがこの構造体のメンバを参照することはまずありません。

ユーザが実際に参照するのは，readdir関数で返されるdirent構造体で，Table 2-4のように定義されています。ユーザはreaddir関数を1回呼び出すごとに，ディレクトリ中で検索したファイルを1つずつ取り出すことができます。またこのとき，d_inoには，stat関数のst_inoと同じ値が代入されます。

**Table 2-2** ● ディレクトリストリーム関数一覧

| 関数名 | 解 説 |
|---|---|
| opendir関数 | ディレクトリストリームをオープンする |
| closedir関数 | ディレクトリストリームをクローズする |
| readdir関数 | 次のディレクトリエントリ構造体のポインタを返す |
| seekdir関数 | ディレクトリストリームの操作位置を変更する |
| rewinddir関数 | ディレクトリストリームの操作位置を先頭に変更する |
| telldir関数 | ディレクトリストリームの操作位置を返す |

**Table 2-3 ● DIR 構造体**

| メンバ名 | 解 説 |
| --- | --- |
| struct _filbuf filesbuf | _dos_files 関数で取得した構造体 |
| long loc | 現在のファイルストリーム位置 |
| long filemax | ディレクトリのファイル数 |
| struct dirent *dp | ディレクトリエントリ構造体へのポインタ |

**Table 2-4 ● dirent 構造体**

| メンバ名 | 解 説 |
| --- | --- |
| ino_t d_ino | 検索したファイルの i-node 番号 |
| char d_name[NAME_MAX + 1] | 検索したファイル名 |

簡単な使用例を List 2-2 に示します。

**List 2-2 ● ディレクトリの内容を表示する**

```
 1:  /*
 2:  ** dirtest.c:
 3:  **    C:ドライブのルートディレクトリに
 4:  **    存在するファイルをすべて表示する
 5:  */
 6:  #include <stdio.h>
 7:  #include <dirent.h>
 8:  #include <errno.h>
 9:
10:  int main (void)
11:  {
12:      /* 構造体の宣言 */
13:      DIR *dirp;
14:      struct dirent *dp;
15:
16:      /* errno の初期化 */
17:      errno = 0;
18:
19:      /* ディレクトリをオープンする */
20:      dirp = opendir ("C:/");
21:      if (dirp == NULL) {
22:          perror ("opendir");
23:          return;
24:      }
25:
26:      /* ファイルが見つからなくなるまで表示する */
27:      while ((dp = readdir (dirp)) != NULL)
28:          printf ("%08x : %s\n", dp->d_ino, dp->d_name);
29:
30:      /* ディレクトリをクローズする */
31:      closedir (dirp);
32:
33:      return 0;
34:  }
```

このプログラムをコンパイルし実行させると，画面には次のように表示されます。

```
A:\> gcc -O dirtest.c
A:\> dirtest.x
02000000 : \HUMAN _Ver3
0200000b : HUMAN.SYS
02000044 : CONFIG.SYS
02000045 : KEY.SYS
02000046 : USKCG.SYS
0200004e : BEEP.SYS
0200004f : STARTUP.ENV
02000050 : COMMAND.X
0200006c : AUTOEXEC.BAT
0200006d : SYS
02000179 : HIS
02000191 : BIN
0200035e : BASIC2
020003c3 : ASK
020003c9 : ETC
A:\> DIR C:\
 \HUMAN _Ver3          C:\
    13 ファイル       1006K Byte 使用中        215K Byte 使用可能
 ファイル使用量          41K Byte 使用
CONFIG             SYS         443  93-02-25  12:00:00
KEY                SYS         712  87-05-15  12:00:00
USKCG              SYS        8028  87-05-15  12:00:00
BEEP               SYS        1023  93-02-25  12:00:00
STARTUP            ENV          33  90-05-15  12:00:00
COMMAND            X         28382  93-02-25  12:00:00
AUTOEXEC           BAT         162  93-02-25  12:00:00
SYS                    <dir>        93-02-25  12:00:00
HIS                    <dir>        93-02-25  12:00:00
BIN                    <dir>        93-02-25  12:00:00
BASIC2                 <dir>        93-02-25  12:00:00
ASK                    <dir>        93-02-25  12:00:00
ETC                    <dir>        93-02-25  12:00:00
A:\>
```

### ◆ インデックスノード

ino または i-node[2] と呼ばれる**インデックスノード**は，ファイルの指標ノードとして **UNIX** のファイルシステムで用いられている概念です。実際，**UNIX** の i-node の情報は複雑で，そのなかには「ファイルのディスク上の位置」,「ファイルの所有者」,「ファイルのアクセス許可」,「アクセス時刻」などが含まれます。

しかし **UNIX** と **Human68k** では，ファイルシステムの概念が異なるため，***LIBC***では **Human68k** のディレクトリエントリを i-node とみなし，疑似的に i-node 番号が重複しないよう，以下の計算式で値を求めています。

**物理ドライブ× 16777216 ＋先頭セクタ番号**

この i-node 番号を用いると，i-node 番号を比較するだけで，簡単にシンボリックリンクなどで同じファイルを指しているかどうかを判定することができます[3]。

2)アイノードと読みます。

3)lstat 関数を用いると，当然 i-node は異なります。

**Table 2-5●拡張UNIXファイルモード設定値**

| マクロ | 意 味 | 所 属 |
|---|---|---|
| S_IXOTH | 第3者実行 | UNIX |
| S_IWOTH | 第3者書き込み | UNIX |
| S_IROTH | 第3者読み込み | UNIX |
| S_IXGRP | グループ実行 | UNIX |
| S_IWGRP | グループ書き込み | UNIX |
| S_IRGRP | グループ読み込み | UNIX |
| S_IXUSR | オーナー実行 | UNIX |
| S_IWUSR | オーナー書き込み | UNIX |
| S_IRUSR | オーナー読み込み | UNIX |
| S_ISVTX | スティッキービット（つねに0） | UNIX |
| S_ISGID | Set GID ビット（つねに0） | UNIX |
| S_ISUID | Set UID ビット（つねに0） | UNIX |
| S_IFIFO | FIFO（つねに0） | UNIX |
| S_IFCHR | キャラクタデバイス | Human68k/UNIX |
| S_IFDIR | ディレクトリ | Human68k/UNIX |
| S_IFBLK | ブロックデバイス | UNIX |
| S_IFREG | 通常ファイル | Human68k/UNIX |
| S_IFLNK | シンボリックリンク | Human68k/UNIX |
| S_IFSOCK | ソケット（つねに0） | UNIX |
| S_IRDONLY | リードオンリー | Human68k |
| S_IFVOL | ボリュームファイル | Human68k |
| S_ISYS | システムファイル | Human68k |
| S_IHIDDEN | 隠しファイル | Human68k |
| S_IEXBIT | 実行ビット | Human68k |

### ◆ ファイルモード

ファイルモードとはHuman68kのファイルアトリビュートのことですが，*LIBC*のchmod関数，stat関数などで用いるファイルモードはUNIXとの互換性を確保するために変更/拡張されています。そのためファイルアトリビュートとは呼ばずに，拡張UNIXファイルモードと呼ぶことにします。ただしDOSコールライブラリでは，*XC*と同じファイルアトリビュートを使用します。

#### 拡張UNIXファイルモードとHuman68kアトリビュートは異なる

拡張UNIXファイルモードの値は，Table 2-5のように定義されています。しかし，拡張UNIXファイルモードを参照する場合は，次のことに注意してください。

また，ファイルモードは下位16ビットこそUNIXのstat関数の返すファイルステータスの内容と同じになりますが，Human68kのファイル構造では完全にUNIXのものとは同一にはできないため，上位16ビットとビットが設定される条件が異なる場合もあります。

- **実行属性/書き込み属性は，オーナー/グループ/第3者すべて同じ値が設定される**
- **読み込み属性はつねに有効である**
- **S_IRDONLY が設定されないかぎり（たとえシステムファイルでも），書き込み属性/読み込み属性はそのまま有効である**
- **実際に実行属性がなくても，拡張子が '.X'，'.R'，'.Z' のものには実行属性がつく**
- **実際に実行属性がなくても，ディレクトリには実行属性がつく**
- **シンボリックリンクファイルは，つねに読み書き属性/実行属性がつく**

## ❖ ファイルの構造

本セクションでは，ディスクマップおよびディレクトリエントリの構造について，ごく簡単に説明しておきます。

### ◆ ディスクマップ

**Human68k** の管理するディスクマップは，デバイスによって領域の大きさやセクタ番号などが変わりますが，おおまかに次のようになっています。

- **IPL プログラム領域**　OS を読み込むために必要なプログラムの領域
- **第1FAT 領域**　データエリアの使用セクタの連鎖（チェイン）状態を示すテーブル
- **第2FAT 領域**　使用されていない
- **ルートディレクトリ領域**　ルートディレクトリのディレクトリエントリの領域
- **データ領域**　実際のファイル内容の領域

### ◆ ディレクトリエントリ

ディレクトリエントリは個々のファイルの情報を記憶しておくためのもので，ルートディレクトリとサブディレクトリのなかに存在します。また，ディレクトリエントリは1ファイルにつき 32 バイトで，Table 2-6 のような構造をもっています。また，ディレクトリエントリに含まれる「ファイルアトリビュート」,「時刻」,「日付」の各フィールドは，それぞれ Table 2-7，Table 2-8，Table 2-9 のようなビット構成になっています。

**Table 2-6●ディレクトリエントリの構造**

| オフセット | 内 容 |
|---|---|
| +$00.B | ファイル名1 (あまった部分は$20) |
| +$08.B | ファイル拡張子 (あまった部分は$20) |
| +$0b.B | ファイルアトリビュート (Table 2-7参照) |
| +$0c.B | ファイル名2 (あまった部分は$00) |
| +$16.W | 時刻 (Table 2-8参照) |
| +$18.W | 日付 (Table 2-9参照) |
| +$1a.W | 先頭FAT番号 (上位/下位が逆) |
| +$1c.L | ファイルサイズ |

**Table 2-7●ファイルアトリビュートのビット構造**

| ビット位置 | ファイルアトリビュート |
|---|---|
| bit 0 | 読み込み専用 |
| bit 1 | 不可視 |
| bit 2 | システム |
| bit 3 | ボリューム |
| bit 4 | ディレクトリ |
| bit 5 | 通常のファイル |
| bit 6 | **Human68k**では未使用だが，フリーウェアの“lndrv.x”が使用する |
| bit 7 | **Human68k**では未使用だが，フリーウェアの“execd.x”が使用する |

**Table 2-8●時刻フィールドのビット構造**

| ビット位置 | 時 刻 |
|---|---|
| bit 15 ～ 11 | 時 |
| bit 10 ～ 05 | 分 |
| bit 04 ～ 00 | 秒×2 |

**Table 2-9●日付フィールドのビット構造**

| ビット位置 | 日 付 |
|---|---|
| bit 15 ～ 09 | 年 |
| bit 08 ～ 05 | 月 |
| bit 04 ～ 00 | 日 |

## ❖ パス名

*LIBC*では，*XC*に比べてパス名の扱いが拡張されており，従来とは扱い方が異なっている場合があります。

**◆ *LIBC*のパス名の扱いについて**

*LIBC*のパス名の扱いは*XC*より柔軟です。*LIBC*では，関数が返すパス名の区切り文字は，一貫して“/”が使用されますから，パス名解析が非常に簡単に行えます。ただし *LIBC*においても，DOS コールライブラリは **Human68k** のバージョンと仕様に依存します。

関数に与えるパス名には，次のような規約があります。

- **ドライブ名は大文字でも小文字でもよい**
- **パス名の区切り文字は“\”または“/”のどちらでもよい**
- **パス名の最後に“\”または“/”が付加されていてもよい**

また関数が返すパス名は，次のような規約にしたがっています。

- **ドライブ名は大文字にする**
- **パス名の区切り文字は“/”とする**
- **パス名の最後に“\”および“/”は付加されない**

# 2.2 ……………………………………浮動小数点演算

## ❖ 数値の表現形式

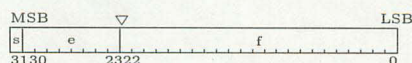
*XC*や **GCC** をはじめとする最近の一般的な C コンパイラがどのようにして実数を扱っているか，その表現形式について説明します。

### ◆ 実数の表現

コンピュータのメモリは有限ですから，無限にある数のうち限られた範囲しか扱えません。このような制限あるコンピュータ上で小数点を表現するために，**固定小数点** (*Fixed point representation*) と**浮動小数点** (*Floting point representation*) という 2 通りの表現方法があります。固定小数点というのは，小数点の位置を固定し，整数部と小数部の桁数をあらかじめ決めて，その範囲内で表現する方法です。また逆に，浮動小数点とは，小数点の位置を指数の形式で記憶しておく方法です。

固定小数点の場合は「扱える数の範囲が狭いが，演算が簡単で速い」のに対し，浮動小数点では逆に「扱える数の範囲は広いが，演算が複雑で固定小数点演算より遅い」という特長があります。そして，これらの 2 つの表現方法のうち，C 言語では実数の表現に浮動小数点を用いています。**GCC** が扱える実数型は，Table 2-10 のとおり[1]です。

### ◆ IEEE 規格の 2 進浮動小数点形式

浮動小数表現にも表現方法はいろいろありますが，現在の一般的な C コンパイラでは，おもに**米国電気電子学会** (*The Institute of Electrical and Electronics Engineers Inc.*) の ANSI/IEEE Std 754-1985(以下，「IEEE [2]」と略す) 規格のフォーマットが採用されています。

1)long double 型もありますが，*X68k Programming Sreies #1 Develop.* に付属している GCC では，double 型と同じ精度として扱われます。

2)「アイ・トリプル・イー」と読みます。

**Table 2-10** ● 各実数型

| 型 | サイズ | 範 囲 | 型の分類 |
|---|---|---|---|
| float 型 | 32 ビット | $1.18 \times 10^{-38} \sim 3.40 \times 10^{38}$ | 単精度浮動小数点 |
| double 型 | 64 ビット | $2.23 \times 10^{-308} \sim 1.80 \times 10^{308}$ | 倍精度浮動小数点 |

float 型 (単精度浮動小数点)

s – 符号部 (0: 正 1: 負)

e – 指数部 (符号なし 8 ビット 2 進整数，ただし e=0，e=0xff は特殊数)

f – 有効数字部 (23 ビット 2 進固定小数点実数)

double 型 (倍精度浮動小数点)

s – 符号部 (0: 正 1: 負)

e – 指数部 (符号なし 11 ビット 2 進整数，ただし e=0，e=0x7ff は特殊数)

f – 有効数字部 (52 ビット 2 進固定小数点実数)

**Fig.2-1● 浮動小数点を構成するビットフィールド**

IEEE フォーマットを利用するメリットは，同じ IEEE フォーマットを使用しているCコンパイラおよびライブラリであれば，機種や OS が異なっていてもつねに同じ結果が得られ，演算精度を一定に保つことができることです。

IEEE 形式の 2 進浮動小数点を構成するビットフィールドは，Fig. 2-1 のように定義されています。このフィールドには「符号 (*sign*)」,「指数 (*exponent*)」,「有効数字 (*significant*) [3)]」の 3 つのフィールドがあり，これらの組み合わせによって実数を表現しています。

3) IEEE 以外では，有効数字を仮数 (*mantissa*) と呼ぶことが多いようです。

また，ビットフィールド中の小数点位置は▽の位置にあり，正規化数は $1 \leq f < 2$ の範囲を表しています。ただし，正規化数の最上位の 1 ビットが省略されているため，有効数字は $1 + f$ となり，実際に格納される $f$ の範囲は $0 \leq f < 1$ となります。これらのフィールドは Table 2-11 のように組み合わされ，次のように分類されます。

○ **非数** ( *Not a number* )

数値的に，無限大より値の絶対値が大きい数を非数 ( *NaN* ) といいます。規定上，非数は有効な値としては扱われず，無効演算の結果として返されます。また，非数についても符号は有効です。

○ **無限大** ( *Infinity* )

演算の結果，値の絶対値が正規化数の最大値より大きくなった場合，その結果を**無限大**といいます。つまり演算がオーバーフローした場合，結果は無限大になります。無限大も 0 と同様に符号が有効で，オーバーフローした値の符号がそのまま用いられます。また，$-\infty < +\infty$ の関係が成り立ちます。

**Table 2-11 ● IEEE 規格の 2 進浮動小数点形式の表現できるフィールドの組み合わせ**

| 表現値 | 符号部 | 指数部 | 有効数字部 |
|---|---|---|---|
| 非数 (NaN) | 0/1 | 111･･･111<br>⋮<br>111･･･111 | 1111･･･1111<br>⋮<br>0000･･･0001 |
| 無限大 (∞) | 0/1 | 111･･･111 | 0000･･･0000 |
| 正規化数 | 0/1 | 111･･･110<br>111･･･110<br>111･･･110<br>⋮<br>000･･･001<br>000･･･001<br>000･･･001 | 1111･･･1111<br>1111･･･1110<br>1111･･･1101<br>⋮<br>0000･･･0010<br>0000･･･0001<br>0000･･･0000 |
| 非正規化数 | 0/1 | 000･･･000<br>⋮<br>000･･･000 | 1111･･･1111<br>⋮<br>0000･･･0001 |
| 0 | 0/1 | 000･･･000 | 0000･･･0000 |
| 単精度 | ←1 ビット→ | ←8 ビット→ | ←23 ビット→ |
| 倍精度 | ←1 ビット→ | ←11 ビット→ | ←52 ビット→ |

○ **正規化数** ( *Normal number* )

**正規化**された浮動小数点のことです。正規化とは，浮動小数点の有効数字のフィールドの最上位の桁が 0 以外の数になるように，指数のフィールドを調整することです。たとえば $+0.000123\times10^{45}$ は，正規化すると $+1.230000\times10^{41}$ になります。

○ **非正規化数** ( *Denormalized number* )

演算の結果，値の絶対値が正規化数の最小値と 0 の間にある数を**非正規化数 (デノーマル数)** といいます。普通，正規化数の最小値より小さい数はアンダーフローとして 0 と認識されてしまいます。しかし IEEE 規格では，アンダーフローが発生した場合にもこのような特殊形を用いて，できるだけ値を失わないようにしています。そのため非正規化数の有効数字フィールドの最上位は，つねに 0 になります。

○ **0** ( *Zero* )

指数/有効数字の両方のフィールドがともに 0 の状態をいいます。IEEE 規格では，0 の場合でも符号は有効で，−0.0 と+0.0 の 2 つの 0 が存在します。これを**符号つき 0** ( *Signed zero* ) といい，−1.0 に近い 0 か，+1.0 に近い 0 かを表しています。ただし，数がもつ量は同じ 0.0 なので，−0.0 = +0.0 であることに変わりはありません。また「真の」0 は，IEEE 規格では+0.0 になります。

## ❖ 数学関数の使用について

*LIBC*の数学関数は，状況に応じて柔軟に動作するように設計されています。最良の結果を得るために，説明をよく読んで使用してください。

### ◆ 数学関数とは

数学関数とは「三角関数」,「対数」,「平方根」など，おもに実数型の演算を行うための関数のことで，<math.h>に定義されています。これらには多くの関数がありますが，*LIBC*には*ANSI C*で規定されるすべての関数がそろっています。

数学関数の簡単な使用例をList 2-3に示します。このプログラムはLC発振周波数をL，Cの定数から求めるものです。

**List 2-3 ● 数学関数の使用例**

```
 1: /*
 2: ** mathtest.c:
 3: **    LC 発振周波数を求めるプログラム
 4: **    f = 1/(2 π * sqrt(LC))
 5: */
 6: #include <stdio.h>
 7: #include <math.h>
 8:
 9: int main(void)
10: {
11:     double l = 1.45, c = 19.4, hz;
12:
13:     printf ("コイル     = %.3f μH\n", l);
14:     printf ("コンデンサ = %.3fpF\n", c);
15:
16:     /* 単位変換 */
17:     l *= 1e-6;
18:     c *= 1e-12;
19:
20:     /* f = 1/(2 π * sqrt(LC)) */
21:     hz = 1 / (M_PI * 2 * sqrt (l * c));
22:
23:     /* 周波数の単位を MHz で表示する */
24:     printf ("周波数     = %.3fMHz\n", hz / 1e6);
25:
26:     return 0;
27: }
```

このプログラムをコンパイルし，実行させせると次のように表示されます。

```
A:\> gcc  -O mathtest.c
A:\> mathtest.x
コイル      = 1.450 μH
コンデンサ = 19.400pF
周波数      = 30.008MHz
A:\>
```

### ◆ *LIBC*の数学関数の特徴

*XC*では，ユーザが“FLOAT2.X”か，あるいは“FLOAT3.X”のどちらのFLOATパッケージ[4)]を組み込むかで，コプロセッサの有無を決定していました[5)]。しかし“FLOAT3.X”などの場合，コプロセッサの処理時間に比べてFLOATパッケージ呼び出しのオーバーヘッドがかなりありました。

*LIBC*の数学関数は，コプロセッサの有無を起動時に自動的に判別します。つまりコプロセッサが実装されていれば，FLOATパッケージを呼び出さずにI/Oコプロセッサおよびコプロセッサ命令を直接使用して，ユーザの環境で最も高速に実行できるようにライブラリ内部の処理を切り替えます。そのため，ある数学関数を使うプログラムをコプロセッサをもっていないユーザが作成したとしても，何ら変更することなくコプロセッサ環境で高速に動作させることができます。

現在**X68000**シリーズのコプロセッサにはインタフェイスの違いにより，アクセス方法が2種類存在します。すなわち，**X68000**機種用に用いられる数値演算プロセッサボードがもつI/O制御によるコプロセッサ方式[6)]と，**X68030**機種に用いられているMPU命令の一部として動作させる方式[7)]です。

**X68000**および**X68030**，コプロセッサの有無といった各環境における動作状態は，Table 2-12のようになります。また，演算速度は一般に以下のような関係になります。

**コプロセッサ命令 ＞ I/Oコプロセッサ ＞ FLOATパッケージ**

4)Human68kのver.3からはコプロセッサ命令による演算を行う“FLOAT4.X”もあります。

5)*XC* ver.2以降では，新たに“FLOATEML.L”ライブラリが追加され，FLOATパッケージなしでソフトエミュレートすることができます。

6)現在“FPPP.X”の仕様により，X68030では作成したコードがキャッシュオン時に動作不安定になるため，X68030では使用不可になっています。

7)X68030の拡張スロットに数値演算プロセッサボードを実装しても，I/Oコプロセッサとして動作するのみで，コプロセッサ命令として動作させることはできません。

### ◆ *LIBC*の数学関数の仕様

*XC*では，エラー処理に*SYSV*相当のmatherr関数を使う方法が用いられていましたが，*LIBC*では，各数学関数のエラーは関数内部で変数errnoに設定するだけにとどめています。このとき，*LIBC*の数学関数で，変数errnoに設定される値はTable 2-13のとおりです。

**Table 2-12** ● 各環境の動作状態

| マシン | コプロセッサ有無 | 動作モード |
|---|---|---|
| **X68000** | なし | FLOATパッケージ呼び出し |
| **X68000** | あり（I/O） | I/Oコプロセッサ |
| **X68030** | なし | FLOATパッケージ呼び出し |
| **X68030** | あり（I/O） | I/Oコプロセッサ（現在は無効） |
| **X68030** | あり | コプロセッサ命令 |

Table 2-13 ● 変数 errno の設定値の意味

| マクロ | 意 味 |
|---|---|
| EDOM | 引数が非数または計算範囲外だった |
| ERANGE | 演算結果がオーバーフローまたはアンダーフローした |

Table 2-14 ● 数学関数のエラー時の戻り値

| 変数 errno の値 | 戻り値 |
|---|---|
| EDOM | 非数 |
| ERANGE | 非正規化数，無限大，非数 |

ただし“FLOAT2.X”を使用した場合，“FLOAT2.X”が非正規化数を返さないという仕様のため，ERANGE が正しく設定されない場合があります[8]。またフリーウェアとして数多く存在する FLOAT パッケージのなかには，高速化のために値のチェック等を省いているものもあり，使用する FLOAT パッケージによっては，結果的に変数 errno が設定されなかったり，正常終了するにもかかわらず変数 errno が設定されることもあります。もし正確な演算をしたければ，コプロッサを用いるか，あるいは純正の“FLOAT2.X”を使用してください。

8) *ANSI C*規格では，エラー番号の ERANGE が設定されるかどうかは機種依存と定義されています。

*ANSI C*規格では一般に正規化数を，また ERANGE エラーの場合は HUGE_VAL[9] を，数学関数の戻り値とするよう定義されていますが，*LIBC*では IEEE 規格にしたがい，可能なかぎり(たとえアンダーフローした数であっても)，その値を返すようにしています。Table 2-14 に *LIBC*数学関数のエラーの戻り値を示します。また，“FLOAT2.X”を使用した場合は非正規化数は返りません。そのことを念頭に置いて使用してください。

9) *4.3BSD*あるいは*SYSV*では，HUGE と定義されていることもあります。

#### ◆ 数学関数の使用について

前述したように，*LIBC*には1つの数学関数に対して「FLOAT パッケージ」，「I/O コプロセッサ直接駆動」，「コプロセッサ命令」の3種類[10]の演算ルーチンが入っています。通常は関数内部でユーザの環境を判断して，それぞれのルーチンを呼び出しますが，さらに少しでも速くしたい場合には直接内部のルーチンを呼び出すことができます。

10) FLOAT パッケージまたはコプロセッサに存在しない数学関数は，内部関数が一部ない場合があります。

この内部ルーチンもほかの標準数学関数と同じくグローバル関数ですので，関数名が異なることとユーザの環境を判断しない部分を除けば，通常の数学関数とまったく同じ動作をします。また，各内部関数の関数名は Table 2-15 のように定義されます。もし，関数名を変えずに簡単に内部関数を使用したいならば，コンパイル時に__DIRECT_FLOAT__ (FLOAT パッケージ呼び出し)，__DIRECT_IOFPU__ (I/O コプロセッサ)，__DIRECT_FPU__ (コプロセッサ命令) の各マクロ[11]のうちいずれか1つを定義してください。それぞれ，内部関数が直接呼び出されます。

11) マクロは<math.h>をインクルードする前に定義してください。

たとえばP.54 の List 2-3 のプログラムを，内部関数呼び出しの形式でコンパイルするには次のようにタイプします。

**Table 2-15** ● 内部ルーチンの関数名

| 動作環境 | マクロ名 | 関数名（???の部分は数学関数名） |
|---|---|---|
| FLOAT パッケージ | __DIRECT_FLOAT__ | _fe_??? |
| I/O コプロセッサ | __DIRECT_IOFPU__ | _fpu_??? |
| コプロセッサ命令 | __DIRECT_FPU__ | _f_??? |

```
A:\> gcc -D__DIRECT_FLOAT__ -O mathtest.c
A:\> gcc -D__DIRECT_IOFPU__ -O mathtest.c
A:\> gcc -D__DIRECT_FPU__ -O mathtest.c
```

しかし，内部関数を直接呼び出すと，ユーザの環境に合わせて動作させることができなくなります。つまり，同一環境のもとでしか動作しなくなります。もしフリーウェアなど，多くの人に配布するようなプログラムを作成するならば，この機能は用いないでください。

## ❖ コプロセッサ補足説明

**X68000** と **X68030** におけるコプロセッサの位置付けについて，簡単に確認しておきます。

### ◆ コプロセッサとは

コプロセッサとは，メインプロセッサのアーキテクチャでは直接サポートされていない命令，レジスタおよびデータタイプをプログラミングモデルに追加するMPU 周辺デバイス LSI です。そのなかの 1 つに数値演算コプロセッサが含まれ，MPU では演算できない多精度の整数演算/ 浮動小数点演算などを MPU の代わりに演算します。

数値演算コプロセッサには，コプロセッサ命令によって MPU 内部の命令の 1 つとして動作するタイプと，I/O を介してプログラムにより制御するタイプの 2 種類があります。MC68000 の標準周辺デバイスのコプロセッサには MC68881 と MC68882 がありますが，この 2 つの LSI は，コプロセッサ命令と I/O による制御の両方で使えるように設計されています[12)]。ただし，コプロセッサ命令は MC68020 以上の MPU でなければ実行できないため，**X68000** では I/O による制御でしか動作させることができません。また，コプロセッサ命令が 1 命令で処理できるのに比べて，I/O コプロセッサの場合，プログラムにより I/O を制御しなくてはならないため，処理が複雑でしかも遅くなります。

12) MC68882 は MC68881 に対して上位互換で，演算速度が速くなっています。ただし，I/O による制御で使う場合は少々異なるようです。

### ◆ MC68881 と MC68882 の判別方法

List 2-4 に MC68881 と MC68882 の簡単な判別方法を紹介しておきます。この 2 つのコプロセッサは，その内部にπなどの定数が書かれた ROM テーブルをもってい

ますが，このテーブルの未定義部分にあたるオフセット$01はMC68881とMC68882では定数が異なり，このテーブルの違いによって判別することができます。コプロセッサ内部ROMの内容は，Table 2-16のようになっていました[13)]。内部ROMは拡張倍精度[14)]の定数なので，double型で表現できないものは16進表示で表記しています。

13)筆者が個人的に調べたデータなので，未定義部分が異なる場合があるかもしれません。

14) IEEE規格のフォーマットでサイズは96ビットですが実際は80ビットデータです。

**List 2-4 ● MC68881とMC68882の判別方法**

```
 1: /*
 2: ** fputest.c:
 3: **   I/Oに実装されているMC68881とMC68882を判別する
 4: */
 5: #include <stdio.h>
 6: #include <sys/dos.h>
 7:
 8: /* インライン展開されると困るので，一応禁止する */
 9: #pragma inline-functions off
10:
11: /* -m68881を指定するからいらないけれど，一応指定する */
12: #pragma 68881 on
13:
14: double getfpurom ()
15: {
16:     double result;
17:
18:     /* 内部ROMのオフセット$01の内容をresultに代入する */
19:     __asm volatile ("fmovecr.x #$01,%0": "=f" (result));
20:
21:     /* オフセット$01の内容を返す */
22:     return result;
23: }
24:
25: /* スーパーバイザ移行前にコプロセッサにI/Oアクセスされないように禁止する */
26: #pragma 68881 off
27:
28: void main(void)
29: {
30:     double offset_1;
31:     int ssp;
32:
33:     /* スーパーバイザに移行する */
34:     ssp = _dos_super (0);
35:
36:     /* オフセット $01 を取得 */
37:     offset_1 = getfpurom ();
38:
39:      /* ユーザに戻る */
40:     if(ssp > 0)
41:         _dos_super (ssp);
42:
43:     /* オフセット内容を判別し，表示する */
44:     printf("OFFSET[$01] = %f : ", offset_1);
45:     if(offset_1 == 0.0)
46:         puts("MC68881です。");
47:     else
48:         puts("MC68882です。");
49: }
```

**Table 2-16 ● コプロセッサ内部 ROM テーブルの内容**

| OFS | MC68881 | MC68882 | 意 味 |
|---|---|---|---|
| $00 | 3.1415926535898 | 3.1415926535898 | $\pi$ |
| $01 | 0.0 | 7.9375031031668 | 未定義 |
| $02 | 7.992189890705 | 7.992189890705 | 未定義 |
| $03 | $200000007FFFFFFF00000000 | $200000007FFFFFFF00000000 | 未定義 |
| $04 | 0 | 0 | 未定義 |
| $05 | 2.2250738585072e − 308 | 2.2250738585072e − 308 | 未定義 |
| $06 | 1.1754942807574e − 038 | 1.1754942807574e − 038 | 未定義 |
| $07 | $00010000F65D8D9C00000000 | $00010000F65D8D9C00000000 | 未定義 |
| $08 | $7FFF0000401E000000000000 | $7FFF0000401E000000000000 | 未定義 |
| $09 | 7.6805736963112e + 304 | 7.6805736963112e + 304 | 未定義 |
| $0A | 6.2307562302418e + 034 | 6.2307562302418e + 034 | 未定義 |
| $0B | 0.30102999566398 | 0.30102999566398 | $log_{10}(2)$ |
| $0C | 2.718281828459 | 2.718281828459 | $e$ |
| $0D | 1.442695040889 | 1.442695040889 | $log_2(e)$ |
| $0E | 0.43429448190325 | 0.43429448190325 | $log_{10}(e)$ |
| $0F | 0.0 | 0.0 | +0.0 |
| $10 | 0.0 | 0.0 | 未定義 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| $2F | 0.0 | 0.0 | 未定義 |
| $30 | 0.69314718055995 | 0.69314718055995 | $ln(2)$ |
| $31 | 2.302585092994 | 2.302585092994 | $ln(10)$ |
| $32 | 1 | 1 | $10^{0}$ |
| $33 | 10 | 10 | $10^{1}$ |
| $34 | 100 | 100 | $10^{2}$ |
| $35 | 10000 | 10000 | $10^{4}$ |
| $36 | 100000000 | 100000000 | $10^{8}$ |
| $37 | 1e + 016 | 1e + 016 | $10^{16}$ |
| $38 | 1e + 032 | 1e + 032 | $10^{32}$ |
| $39 | 1e + 064 | 1e + 064 | $10^{64}$ |
| $3A | 1e + 128 | 1e + 128 | $10^{128}$ |
| $3B | 1e + 256 | 1e + 256 | $10^{256}$ |
| $3C | $46A30000E319A0AEA60E91C7 | $46A30000E319A0AEA60E91C7 | $10^{512}$ |
| $3D | $4D480000C976758681750C17 | $4D480000C976758681750C17 | $10^{1024}$ |
| $3E | $5A9200009E8B3B5DC53D5DE5 | $5A9200009E8B3B5DC53D5DE5 | $10^{2048}$ |
| $3F | $75250000C46052028A20979B | $75250000C46052028A20979B | $10^{4096}$ |

しかし，この方法は未定義部分を参照しているので，将来LSIチップ自体が変更されて使用できなくなるかもしれません。そのことには注意してください

このプログラムをI/Oコプロセッサ(この場合，MC68881)を実装した状態で実行すると，次のようになります。

```
A:\> gcc -O -m68881 fputest.c -ldos
A:\> fputest.x
OFFSET[$01] = 0.000000 : MC68881です。
A:\>
```

ただしList 2-4はI/Oコプロセッサの判別だけで，I/Oコプロセッサが実装されているかどうかについては調べていません。さらに，このプログラムは少々危険なコーディングで，*X68k Programming Sreies #1 Develop.* に付属のGCCでしかコンパイルすることができません。もっと簡単に判別するために，*LIBC*では_is68881関数を提供していますから，こちらの使用をお勧めします。

# 2.3 ………………………………………… 日付と時間

## ❖ 暦時間

本セクションでは時間の表現方法，協定世界時と地域時間の違いなど，*LIBC*での日付と時間の取り扱いについて簡単に解説します。

### ◆ 暦時間とは

コンピュータ上で日付や時間を表現する方法は，実は多種多彩で，OS によってあるいはライブラリによって異っています。しかし一般には，**暦時間**というものが使われることが多いようです。

暦時間というのは，英語で**カレンダータイム** (*Calendar time*) あるいは**エポックタイム** (*Epoch time*) といいます。これらは，**協定世界時**である 1970 年 1 月 1 日午前 0 時 0 分 0 秒[1]から何秒経過したかを，long 型で表したものです。ただし，この 1970 年という基点が曲者で，処理系によっては 1900 年，場合によっては 1980 年となっていたりして，混乱しています[2]。このような基点のズレはありますが，その意味はいずれも処理系依存の**エポック**から何秒経過したかということですから，大した問題ではありません。

1)この時刻をエポック (英語では epoch) と呼びます。

2)そのなかでは，1970 年に設定している場合が多いようです。

むしろ注意すべきなのは，暦時間と**地域時間**には関係がないということです。このことは勘違いしやすいことですから，はっきりさせておきましょう。繰り返しますが，暦時間は「協定世界時」で計算した 1970 年 1 月 1 日からの経過秒数のことであり，自分が住んでいる地域での地域時間 (たとえば日本時間) で計算した 1970 年 1 月 1 日からの経過秒数ではありません。

**暦時間はつねに協定世界時である**

### ◆ 暦時間を表現する

暦時間は，C では<time.h>で定義される time_t 型で表現します。ただし，*LIBC*では time_t 型は long 型への typedef ですから，表現できる値は $\pm 2^{31}$ までであり，近い将来，おそらく 21 世紀半ば[3]には long 型で表現できなくなります。しかしそのような将来に，いまだに現在と同じ形で計算機を用いていると考えるのは，実にばかばかしいことなので問題とはならないでしょう。

3)もし暇な人がいたら，実際に何年何月何日になるか計算してみましょう。

◆ 今の暦時間を求める

今現在の暦時間を求めるには，time 関数を使います。ほかに暦時間を求める方法として ftime 関数がありますが，こちらはどちらかといえば古くさいインタフェイスですから，ここでは解説しません。必要ならば Vol.2 のマニュアルを参照してください。

**List 2-5 ● time 関数の使い方**

```
 1: /*
 2: ** time.c:
 3: **    どちらの関数も今現在の暦時間を求める
 4: */
 5: #include <time.h>
 6:
 7: time_t getCalendarTime1 (void)
 8: {
 9:     /* NULL を指定するのが普通 */
10:     return time (NULL);
11: }
12:
13: time_t getCalendarTime2 (void)
14: {
15:     time_t now;
16:
17:     /* さもなければこういう使い方もできる */
18:     time (&now);
19:     return now;
20: }
```

◆ 任意の暦時間を求める

では，任意の日付/時間の暦時間を求めるにはどうすればよいでしょうか。そのためには mktime 関数を使います。mktime 関数は，**詳細時間**（tm 構造体で表現）を time_t 型に逆変換するための関数です。詳細時間については後ほど詳しく述べますので，ここではサンプルプログラムを示すだけにとどめておきましょう（List 2-6 参照）。ただし，mktime 関数に指定する時間は地域時間で，協定世界時ではありません。このことには注意してください。

**mktime 関数には地域時間を指定する**

**List 2-6 ● mktime 関数の使い方**

```
 1: /*
 2: ** mktime.c:
 3: **    日本時間 1993 年 2 月 10 日 10 時 0 分 0 秒を暦時間に直す
 4: */
 5: #include <time.h>
 6:
 7: time_t convertTime (void)
 8: {
 9:     struct tm t;
10:
11:     t.tm_sec    = 0;    /* 秒 */
```

```
12:     t.tm_min   = 0;   /* 分 */
13:     t.tm_hour  = 10;  /* 時 */
14:     t.tm_mday  = 10;  /* 日 */
15:     t.tm_mon   = 1;   /* 月 - 1 */
16:     t.tm_year  = 93;  /* 年 - 1900 */
17:     t.tm_isdst = -1;  /* 夏時間を考慮 */
18:     return mktime (&t);
19: }
```

### ◆ 暦時間の利点

暦時間を用いると，2つの時間の差が簡単に求められます。たとえば暦時間を使えば，1992年10月3日5時17分30秒と1993年1月15日8時30分0秒という複雑な時間差は，単なる引き算1つで得られます。これをばか正直に秒/分/時の順番で計算するとしたら，演算ルーチンはおそらく数十行以上のものになるでしょうし，バグの温床になるのは明白です[4)]。

4)加えていうならば，頭痛薬も必要でしょう。

ただし，安易に引き算で計算できるかといえばそうではありません。先ほど引き算1つで求まると述べたのは，あくまでも*LIBC*で`time_t`型を`long`型として定義しているからにすぎないからです。もし`time_t`型が構造体で定義されているような処理系では，問題となるでしょう。事実*ANSI C*では，`time_t`型をどのように表現するかについては定義していません。

ですから，*ANSI C*に準拠した行儀のよい，移植性の高いプログラムを書くためにも，`difftime`関数を使うべきです。*LIBC*では，`difftime`関数は引き算に変換されるマクロとして定義されていますから，速度的にも差はありません。むしろソースの移植性を重視すべきです。

**List 2-7 • 2つの時間の差**

```
 1: /*
 2: ** difftime.c:
 3: **   1992年10月3日5時17分30秒と
 4: **   1993年1月15日8時30分0秒の差を求める
 5: */
 6: #include <time.h>
 7:
 8: double getDifference (void)
 9: {
10:     time_t c1, c2;
11:     struct tm t1, t2;
12:
13:     t1.tm_sec   = 30; t2.tm_sec   = 0;  /* 秒 */
14:     t1.tm_min   = 17; t2.tm_min   = 30; /* 分 */
15:     t1.tm_hour  = 5;  t2.tm_hour  = 8;  /* 時 */
16:     t1.tm_mday  = 3;  t2.tm_mday  = 15; /* 日 */
17:     t1.tm_mon   = 9;  t2.tm_mon   = 0;  /* 月 - 1 */
18:     t1.tm_year  = 92; t2.tm_year  = 93; /* 年 - 1900 */
19:     t1.tm_isdst = -1; t2.tm_isdst = -1; /* 夏時間を考慮 */
20:
21:     c1 = mktime (&t1);
22:     c2 = mktime (&t2);
23:     return difftime (c2, c1);
24: }
```

## ❖ 協定世界時

時間には，地域時間と協定世界時の２通りの表現方法があります。本セクションでは，そのうちの協定世界時について解説します。

### ◆ 協定世界時とは

**協定世界時**はいくつかある**世界時**のうちの１つで，**セシウム原子**の放射から算出された**原子時**に対して，**閏秒**などの修正を加えて地球の自転に合致させたものです。英語では *Coordinated Universal Time* (略称 UTC) といい，1925年1月1日から**グリニッジ標準時** (*Greenwich Mean Time* 略称 GMT) に代わって用いられています。

## ❖ 地域時間

前セクションで協定世界時について解説しました。ここではもう一方の表現方法である，地域時間について解説します。

### ◆ 地域時間とは

**地域時間**とは，**協定世界時**に対して**時間帯** (**タイムゾーン**) による時差を補正したもので，いわゆる**日本時間**がこれにあたります。日本は+9時間のタイムゾーンに属しており，地域時間は協定世界時より9時間進んでいることになります。

また，地域時間は時差の補正だけではなく，国ごと，あるいは地域ごとに制定された**夏時間**などの補正が適用されることがあります。今のところ，日本では採用されていませんが，アメリカなどでは夏時間 (*Daylight Saving Time*：略称 DST)[5] を採用しています。

これは１年のうちのある期間だけ時間をずらし，季節によって異なる日の長さを有効に利用しようというもので，１時間ほど時間を進めるのが通例のようです。

5) 日本ではサマータイム制などと呼ばれることもあります。

### ◆ 地域時間の見分けかた

コンピュータが協定世界時から地域時間を計算するときに必要な情報は，最低限必要なもの，あってもなくてもいいものの両方を含めると，およそ次のようになります。

- **協定世界時からの時差**
- **平常時の時間帯名 (たとえば日本は JST)**
- **夏時間の実施スケジュール**
- **夏時間の実施期間中の時差**
- **夏時間の実施期間中の時間帯名**

これらの情報から，適切に地域時間を計算しなくてはならない点ではいずれの処理系も同じですが，その情報の取得方法という側面からは処理系ごとに異なっているようです。たとえば **UNIX** では，大別して 2 種類あるように見うけられます。

1. **環境変数 TZ から地域時間名を取得して地域時間データベースを検索**
   たとえばソニーの NEWS–OS はこの方法をサポートしています。日本で使うならば，`TZ=Japan` などとします。
2. **環境変数 TZ にすべての情報をつめ込む**
   この方法は ***POSIX.1*** で規定されており，上記のすべての事項について記述できるようになっています。たとえば日本で使うならば，`TZ=JST-9` などとします。

**UNIX** では血筋でしょうか[6]，こういった地域時間への対応がなされているものがほとんどですが，パソコン上の **MS–DOS** や **Human68k** などは地域時間しか考えていないものが多いようです。

6)アメリカなどは同じ国のなかで，いくつもの時間帯に分割されていますから必然的にそうなってきたのではないでしょうか。

***LIBC*** では上述した ***POSIX.1*** スタイルのタイムゾーン処理をサポートしていますが，こういった **Human68k** の仕様上の問題で，どうしても正確な計算ができません。これは **MS–DOS** にしても同じなのですが，OS を通して扱う時間がすべて地域時間であることです。**UNIX** では OS で扱う時間は暦時間，つまり協定世界時ですから，世界中どこでもタイムゾーンの設定さえすれば使用できます。しかし，OS が地域時間しか扱えないとなると，時間帯をまたがった場合にいろいろな問題が生じてきます。

たとえば，ある人が日本で仕事をし，“`FILE.1`” を作成したとします。このファイルは 1993 年 3 月 1 日正午，つまり協定世界時の 1993 年 3 月 1 日午前 3 時に作成したもので，それ以降は何もしていません。

```
A:\> DIR
ドライブ A: のボリュームラベルは HUMAN
ディレクトリは A:\

BIN           <DIR>    93-03-01    08:30:00
FILE     1      5000   93-03-01    12:00:00
FILE     2      4872   93-01-01    14:23:00
        4 個のファイルがあります。
 20259568 バイトが使用可能です。
```

次に，その人は協定世界時からの時差が 2 時間の国へ，そのディスクをもっていきます。その国で同じようにしてディスクの中身を見ると，おそらく次の画面のように表示されるでしょう。なぜならばコンピュータの時間設定が，OS 上でその国の時間に合わされているからです。

```
A:\> DIR
ドライブ A: のボリュームラベルは HUMAN
ディレクトリは A:\

BIN          <DIR>      93-03-01   08:30:00
FILE    1         5000  93-03-01   12:00:00
FILE    2         4872  93-01-01   14:23:00
        4 個のファイルがあります。
  20259568 バイトが使用可能です。
```

つまり，時間がすべて地域時間で記録されているために，どこへもっていっても同じ時間になってしまうのです。考えてみればこの時間帯+2時間の国で，1993年3月1日正午に作成したと表示されているわけですから，“FILE.1”は協定世界時の1993年3月1日午前10時に作成されたということになってしまいます。ここで7時間ものずれが生じてしまい，事実が失われてしまいました。

しかし正しく協定世界時と地域時間を使い分けるOSさえ使っていれば，このような事態にはなりません。今の例と同じようなことを，**UNIX**上で実験してみましょう。まず，日本で見た結果。

```
% ls -lF
total 2
drwxr-xr-x 2 mura        512 Mar  1 08:30 BIN/
-rw-r--r-- 1 mura       5000 Mar  1 12:00 FILE.1
-rw-r--r-- 1 mura       4872 Jan  1 14:23 FILE.2
```

次に時差2時間の国で見た結果。もちろん正しくタイムゾーンを設定しています。

```
% ls -lF
total 2
drwxr-xr-x 2 mura        512 Mar  1 01:30 BIN/
-rw-r--r-- 1 mura       5000 Mar  1 05:00 FILE.1
-rw-r--r-- 1 mura       4872 Jan  1 07:23 FILE.2
```

正しく時間が表示されました。一見すると時間が変わってしまったように思いますが，協定世界時に直してみれば，同じ時間を表していることに気づくはずです。ここが地域時間の重要な部分です。

### 地域時間を基準としてはならない

#### ◆ *LIBC*での制限事項

*LIBC*では，可能なかぎり正確に協定世界時と地域時間を区別して扱うように努めていますが，上記のような理由から，ある程度制限されてしまいます。そのため，*LIBC*で「正確に」時間を扱う場合には，次のことに注意してください。

1. **日本で使うこと**

   **Human68k** がシステムクロックを地域時間でしか扱えない以上，それを協定世界時に逆変換するには仮定が必要です。つまり，「今使っているのは日本であり，システムクロックは日本時間を表している」ということです。*LIBC* は，**Human68k** から得た現在時刻から協定世界時を求めるために，つねに9時間を引いています。しかし日本以外，正確には+9時間の時間帯外では，先ほど述べたような問題が残っています。

2. **フルデコード tzset 関数を使うこと**

   *LIBC*には，タイムゾーンを計算するための tzset 関数が2種類あります。1つは簡易型で，日本時間しか扱えず夏時間についても考慮しません。もう1つは完全型で，*POSIX.1* で規定されている環境変数 TZ の設定をすべてサポートしており，将来日本で夏時間が導入されても，環境変数の設定だけで正しく時間を扱うことができます。ただし，完全型は実行コストがかかるうえに実行ファイルのサイズが大きくなってしまいますから，デフォルトでは簡易版が使用されるようになっています。もし完全型を使用したければ，コンパイラに -ltz を指定するか，リンカに直接 libtz.a を指定するようにしてください。

```
A:\> gcc -O jikan.c -ltz
```

## ❖ 詳細時間

ここまでで暦時間/協定世界時/地域時間について解説してきました。おおまかな概念はつかめたでしょうか。それでは，これらの時間を実際に扱ってみることにしましょう。

### ◆ 詳細時間とは

**暦時間**がいわば，計算向きの表現方法であるのに対して，**詳細時間**は年月日/時分秒を独立して扱うことのできる人間向きの表現方法といえます。詳細時間は，tm 構造体で表現されています。tm 構造体は，それぞれ Table 2-17 のようなメンバで構成されています。

### ◆ tm 構造体の再計算

この tm 構造体のメンバのうち tm_wday, tm_yday については，先ほど説明した mktime 関数 (List 2-6 参照) を使って，tm 構造体から time_t 型に逆変換するときには設定しなくてもかまいません。なぜならば，これらの構造体メンバはほかの値から計算できるからです。逆にいえば，これらの値を求めるために mktime 関数を用いることができます (List 2-8 参照)。

Table 2-17 ● tm 構造体の内容

| メンバ名 | 範囲 | 解説 |
|---|---|---|
| int tm_sec | $0 \leq x \leq 59$ | 秒 |
| int tm_min | $0 \leq x \leq 59$ | 分 |
| int tm_hour | $0 \leq x \leq 23$ | 時 |
| int tm_mday | $1 \leq x \leq 31$ | 日 |
| int tm_mon | $0 \leq x \leq 11$ | 月から1を引いたもの |
| int tm_year | $0 \leq x$ | 年から1900を引いたもの |
| int tm_wday | $0 \leq x \leq 6$ | 曜日 (日曜日を0とする) |
| int tm_yday | $0 \leq x \leq 365$ | 年の通算日 |
| int tm_isdst | 0 *or* 1 | 夏時間の期間中は1 |
| char *tm_zone | | 地域時間名 (JST など) |
| long tm_gmtoff | | 協定世界時と地域時間の差 (秒) |

List 2-8 ● mktime 関数の別の使い方

```
 1: /*
 2: ** mktime2.c:
 3: **    1985年12月14日は何曜日だったかを求める。もちろんもっと
 4: **    簡単な方法があるが, 思い出せなければこういう方法でも可
 5: **    能ということである
 6: */
 7: #include <time.h>
 8:
 9: int getWeekDay (void)
10: {
11:     struct tm t;
12:
13:     t.tm_sec   = 0;    /* 秒 */
14:     t.tm_min   = 0;    /* 分 */
15:     t.tm_hour  = 0;    /* 時 */
16:     t.tm_mday  = 14;   /* 日 */
17:     t.tm_mon   = 11;   /* 月 - 1 */
18:     t.tm_year  = 85;   /* 年 - 1900 */
19:     t.tm_isdst = -1;   /* 夏時間を考慮 */
20:     mktime (&t);       /* mktime が足りない部分を補う */
21:     return t.tm_wday;
22: }
```

### ◆ 協定世界時と地域時間

協定世界時や地域時間を求めるには, gmtime 関数や localtime 関数を使いますが, どちらも結果として, この tm 構造体へのポインタを返してきます。ただし, 注意点があります。それはこれらの関数, つまり gmtime 関数と localtime 関数の返してくるポインタが関数内部の静的領域を指しているということ, 要するに, 関数をもう一度呼び出すと前の値が消えてしまうということです。List 2-9 に正しい例と失敗例を示してみます。

#### tm 構造体の内容は上書きされる

**List 2-9 ● gmtime 関数，localtime 関数の使い方**

```
 1: /*
 2: ** localtime.c:
 3: **   現在の協定世界時と地域時間を調べる悪い例
 4: */
 5: #include <time.h>
 6:
 7: void printTime1 (void)
 8: {
 9:     time_t now;
10:     struct tm *tgmt;
11:     struct tm *tloc;
12:
13:     /*
14:     ** gmtime の結果を参照する前に localtime で結果を
15:     ** 上書きしてしまっている。どちらも JST を表示する
16:     */
17:     now = time (NULL);
18:     tgmt = gmtime (&now);
19:     tloc = localtime (&now);
20:
21:     printf ("UTC time is %02d:%02d:%02d\n",
22:             tgmt->tm_hour, tgmt->tm_min, tgmt->tm_sec);
23:
24:     printf ("JST time is %02d:%02d:%02d\n",
25:             tloc->tm_hour, tloc->tm_min, tloc->tm_sec);
26: }
27:
28: /*
29: ** 現在の協定世界時と地域時間を調べるよい例
30: */
31: void printTime1 (void)
32: {
33:     time_t now;
34:     struct tm *tgmt;
35:     struct tm *tloc;
36:
37:     now = time (NULL);
38:
39:     /*
40:     ** tm 構造体のコピーを行いたくないならば，その場その場で
41:     ** 値を参照すること
42:     */
43:     tgmt = gmtime (&now);
44:     printf ("UTC time is %02d:%02d:%02d\n",
45:             tgmt->tm_hour, tgmt->tm_min, tgmt->tm_sec);
46:
47:     tloc = localtime (&now);
48:     printf ("JST time is %02d:%02d:%02d\n",
49:             tloc->tm_hour, tloc->tm_min, tloc->tm_sec);
50: }
```

### ◆ 文字列に直す – その 1 –

詳細時間を文字列に直すにはいくつかの方法があります。そのなかで最も簡単なのは，asctime 関数と ctime 関数を使う方法です。これらの関数は，与えられた tm 構造体 へのポインタから，決められたフォーマットで文字列に直してくれ

ます。ただし，どちらも指定するのは協定世界時でなければいけません。それを asctime 関数は協定世界時で，ctime 関数は地域時間で表現します。

**asctime 関数，ctime 関数には協定世界時を与える**

**List 2-10 ● asctime 関数と ctime 関数**

```
 1: /*
 2: ** jikan.c:
 3: **    現在時刻を求め，それを協定世界時と地域時間で表示する
 4: */
 5: #include <stdio.h>
 6: #include <time.h>
 7:
 8: int main (void)
 9: {
10:     time_t now;
11:     struct tm *tmptr;
12:
13:     now = time (NULL);       /* 現在時刻を得て */
14:     tmptr = gmtime (&now);   /* 協定世界時の詳細時間に変換する */
15:
16:     fputs (asctime (tmptr), stdout); /* 協定世界時を表示 */
17:     fputs (ctime (tmptr), stdout);   /* 地域時間を表示 */
18:
19:     return 0;
20: }
```

このプログラムを実行させると，次のような結果を出力します。

```
A:\> jikan
Tue Mar  1 23:24:54 1993
Tue Mar  2 08:24:54 1993
A:\>
```

### ◆ 文字列に直す – その2 –

決まったフォーマットでよいならば，先ほどの asctime 関数や ctime 関数が便利ですが，もちろん自分の好きなフォーマットで日付を出力する方法もあります。strftime 関数を使うと，詳細時間を自分の好きなように加工/出力することができます。ただし，strftime 関数は詳細時間を文字列に変換して加工するだけなので，協定世界時と地域時間の変換はあらかじめ施しておく必要があります。

**strftime 関数を呼ぶ前に地域時間に変換すべし**

詳しいフォーマットは Vol.2 のマニュアルを参照していただくとして，いくつか変わったフォーマットを定義して表示させてみましょう (List 2-11 参照)。すべて現在時刻を表示しています。

**List 2-11 ● strftime 関数の使い方**

```
 1: /*
 2: ** hyouji.c:
 3: **   strftime 関数を使った表示例
 4: */
 5: #include <stdio.h>
 6: #include <time.h>
 7:
 8: int main (void)
 9: {
10:     time_t now;
11:     struct tm *tmptr;
12:     char buffer[256];
13:
14:     now = time (NULL);         /* 現在時刻を得る */
15:     tmptr = localtime (&now); /* 地域の詳細時間に変換する */
16:
17:     /* 例 1 */
18:     strftime (buffer, sizeof buffer, "%x %X %Z", tmptr);
19:     puts (buffer);
20:
21:     /* 例 2 */
22:     strftime (buffer, sizeof buffer, "%A %B %d %Y (%p %I:%M:%S)", tmptr);
23:     puts (buffer);
24:
25:     /* 例 3 */
26:     strftime (buffer, sizeof buffer, "%a %d/%m/%y [%j]", tmptr);
27:     puts (buffer);
28:
29:     return 0;
30: }
```

上記のプログラムをコンパイルし実行させると，画面には次のように表示されます。

```
A:\> hyouji
03/02/93 14:02:20 JST
Tuesday March 02 1993 (PM 02:02:20)
Tue 02/03/93 [061]
A:\>
```

# 2.4 ヒープ領域

## ❖ ヒープ領域の実際

ヒープ領域とは，C言語で作成されたプログラムが自由に利用できるメモリ領域のことです。このメモリ領域からは，malloc関数によって指定したサイズのメモリブロックを切り出したり，逆にfree関数によって戻したりすることができます。本セクションでは，ヒープ領域の取り扱いについて解説します。

### ◆ ヒープ領域とは何か

よく勘違いされやすいのですが，ヒープ領域は**Human68k**が管理しているメモリ領域ではありません。**Human68k**がプログラムに対して与えているメモリブロックのなかの一部分で，C言語のライブラリが管理するメモリ領域です。この関係を図に示すと，Fig. 2-2，Fig. 2-3のようになります。システム全体のメモリ空間に対して，Fig. 2-2は**Human68k**が管理している[1)]メモリブロックの並び方を図示したものであり，Fig. 2-3はそのなかの「現在実行中のプログラム」の部分をさらに拡大したものです。

1)Human68kのメモリブロックを取得したり解放するのはDOSコールで行います。

**ヒープ領域は実行中のプログラムの一部である**

### ◆ ヒープ領域の拡張

ヒープ領域は有限ですから，malloc関数によってメモリブロックを確保していくと，そのうち空き領域がなくなってしまいます。このとき，Cライブラリはヒープ領域をsbrk関数によって拡張しようとし，sbrk関数は**Human68k**に対して自分のプログラムが占めている**Human68k**のメモリブロックを大きくしてくれるようにDOSコールSETBLOCKを発行します。

もし，ここでFig. 2-2中の「プロセスの後ろの空き領域」に余裕があれば，**Human68k**はこの要求を認め，プログラムが占めているメモリブロックを拡張します。それによって，Fig. 2-3中のヒープ領域のサイズも大きくなりますから，新たにmalloc関数によって割り当てられる領域が増え，万事めでたしということになります。

しかし「プロセスの後ろの空き領域」がなく，すぐ後ろに**Human68k**が管理する別のメモリブロックがあったり，物理メモリ一杯まで使い切ってしまうと，

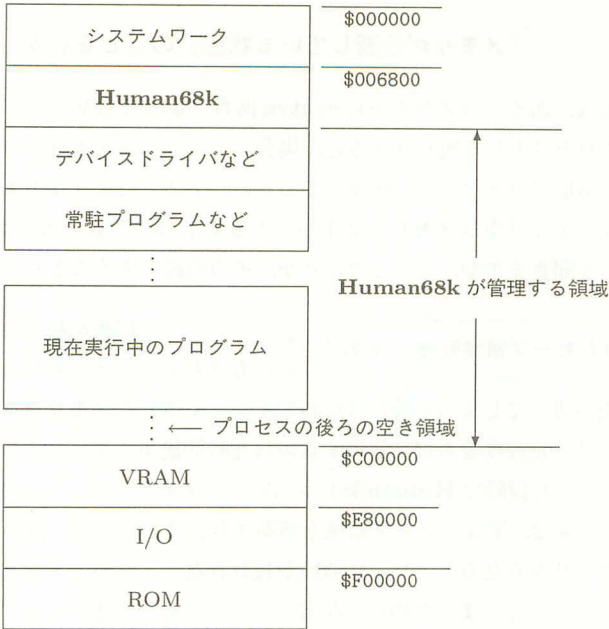



**Fig.2-2●** システム全体と Human68k の管理するメモリの関係

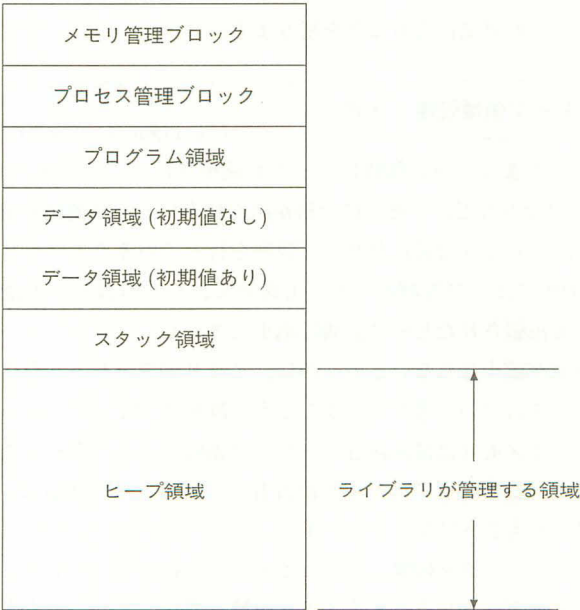



**Fig.2-3●** プログラム内部のヒープ領域の位置

2)変数 errno に設定されるエラー番号は ENOMEM です。

**Human68k** はプログラムが占めているメモリブロックを拡張することができませんから，連鎖的に malloc 関数も新しいメモリブロックの割り当てができなくなります。これがC言語のライブラリでいうところの「**メモリ不足**」2)です。

### 「メモリが分断している状態」のことをいう

たとえば，あるプログラムから system 関数によって常駐型のソフトウェアなどを子プロセスとして実行させると，場合によっては現在実行中のプロセスの後ろにその常駐ソフトウェアが居座ってしまい，上述したような現象になることがあります。よく「空きメモリが 2M バイトもあるのに空きメモリがなくなった」という話を聞きますが，こうしたことが，その原因であることも多いようです。

#### ◆ 優れたヒープ領域管理 – その1 –

上記のような問題が起きないヒープ管理の方法もあります。もちろん物理メモリを全部使い切ってしまった場合はだめですが，上述した**メモリブロックの分断**によるメモリ不足程度ならば，解決するのは比較的簡単です。ヒープ領域の拡張ができなくなった段階で **Human68k** から新しいメモリブロックを割り当ててもらい，そこに第2，第3のヒープ領域を構築すればよいのです。確かに管理する側のライブラリから見ると，ヒープ領域が複数存在し，しかもメモリ空間のなかで飛び飛びになってしまうために，非常にやりにくくなりますが，アプリケーション側から見るかぎりはその違いがわからないので，有用な方法だと考えられます。

3)*LIBC* では Boundary Tag Method という動的メモリ管理方法をベースとして，少し改良を加えたものを使用しています。

しかし残念ながら，*LIBC* のヒープ領域管理ルーチン3)は同時に1つのヒープ領域しか扱えませんから，このような芸当はできません。複数のヒープ領域を考えるときに，sbrk 関数や brk 関数の関数の仕様を満足するにはどのようにするか，その解決方法がまだ煮つまっていないためです。将来，バージョンアップによってこのことが可能になることを祈ります。

#### ◆ 優れたヒープ領域管理 – その2 –

一般に，ヒープ領域はメモリ不足が起こるたびに自動的に拡張されていきますが，逆にメモリがあまっている状態になっても縮小されることはありません。*GNU malloc* ライブラリなど，メモリに余裕が出た場合にヒープ領域を縮小するものもありますが，このような優れたヒープ管理を行っているライブラリは小数派に属します。*LIBC* のヒープ管理ルーチンも含めて，むしろほかの大他数のライブラリでは，一度拡張されたヒープ領域は縮小しません。

ヒープ領域が縮小されないというのは，メモリ効率をひどく悪化させます。たとえばエディタについて考えてみましょう。最初に 1M バイト近いファイルを編集しようとしてメモリに読み込むと，ヒープ領域はそれに合わせてかなり拡張されるはずです。この時点でエディタが占有しているメモリ領域は，1M バイトをはるかに越える大きさになるでしょう。しかし，その後ファイルの編集が終了し，そのファイルのデータを破棄したとしても，一度拡張されたヒープ領域は縮小されず，またプログラムが占有するメモリ領域も変化しません。本来，それだけのヒープ領域を必要としていなくても，です。

前述した複数ヒープ領域の管理やヒープ領域の自動縮小などの課題が，*LIBC*には残されています。おそらくは管理方法をかなり変更しないと，実現するのは難しいかもしれません。

## ❖ 配列と動的メモリ確保

まず最初に，配列と動的メモリ確保 (*Dynamic Memory Allocation*) の違いから考えてみましょう。本質的には，どちらもメモリ上のある領域をアクセスするためのものであり，大きな違いはないのですが，それぞれに長所/短所があります。

### ◆ 配列の長所

最初に，配列の長所から説明しましょう。配列では宣言時に初期値が明確に決定されます。つまり，配列に対して初期化要素を与えればその初期化要素が初期値になり，なければ0が初期値となります。そしてC言語で規定される変数のスコープにしたがって管理されるため，メモリ領域の確保/解放などに気を使う必要がありません。また配列は宣言することも，扱うことも非常に簡単です。特に，多次元 (2, 3, . . . , $n$ 次元) のデータを扱うには，配列を使うのがいちばん簡単です。たとえばList 2-12がその一例です。

**List 2-12 ● 配列の長所**

```
 1:  /*
 2:  ** array.c:
 3:  **   配列は初期化要素がすべて 0 である。また多次元のデータに容易に
 4:  **   アクセスできる
 5:  */
 6:  int array[50][50]; /* 2 次元配列 */
 7:
 8:  void InitializeArray (void)
 9:  {
10:      int i;
11:
12:      /*
13:      ** 配列の座標 (n,n) の値を n にする。それ以外の座標の値は暗黙の
14:      ** うちに 0 であると期待してもかまわない。また，2 次元配列へのアクセ
15:      ** スも array[i][i] のように非常に簡単である
16:      */
17:      for (i = 0; i < 50; i++)
18:          array[i][i] = i;
19:  }
```

### ◆ 配列の短所

配列のサイズが宣言時に決定されてしまうということは，逆にいえば後から変更することができないということです。たとえば要素数が1000の配列を宣言したとすると，実際に必要な要素数が100であれば残り900はムダになるし，2000

であれば1000も足りません。つまり固定サイズの配列は，状況に応じて最適なサイズに調整することができないのです。このため，えてして配列の要素数がプログラムの機能の制限を生み出すことになります。たとえばList 2-13がその一例です。

**List 2-13 ● 配列の短所**

```
 1:  /*
 2:  ** array_str.c:
 3:  **    配列を用いると，状況の変化に対して柔軟に対処すること
 4:  **    ができない。文字列の入力を例にとると....
 5:  */
 6:  #include <stdio.h>
 7:
 8:  char *getString (void)
 9:  {
10:      static char buffer[256]; /* 256バイトの文字配列 */
11:
12:      /*
13:      ** fgets 関数は配列を用いるので，この場合256バイトを
14:      ** 越える文字列を処理することができない
15:      */
16:      return fgets (buffer, sizeof buffer, stdin);
17:  }
```

### ◆ 動的メモリ確保の長所

次に，動的メモリ確保の長所について説明しましょう。必要に応じて必要なサイズのメモリ領域を確保することができ，しかもそれを配列のように自由に扱うことができます。また，いったん領域確保した後でも，そのサイズを変更することができます。したがって，状況に応じて柔軟に対処できるプログラムを作成することができます。

たとえば次のList 2-14のようなプログラムを，配列を用いて作成したとします。大きなファイルも読み込めるようにと，1Mバイトの配列を取ってしまうと，10Kバイト程度の小さなファイルでも1Mバイトのメモリが必要になる反面，2Mバイトのファイルを読み込むことはできません。

**List 2-14 ● 動的メモリ確保の長所**

```
 1:  /*
 2:  ** malloc.c:
 3:  **    動的メモリ確保は状況に応じて領域のサイズを動的に変更すること
 4:  **    ができるのが強み。たとえば次の例は，ファイルの中身をメモリに
 5:  **    読み込む処理だが，ファイルの大きさによって読み込み領域のサイ
 6:  **    ズを決定している
 7:  */
 8:  #include <stdio.h>
 9:  #include <stdlib.h>
10:  #include <sys/stat.h>
11:  #include <unistd.h>
12:  #include <fcntl.h>
13:
14:  unsigned char *file_contents; /* ファイルの中身を読み込む領域 */
```

```
15:
16: void readFileIntoMemory (char *file_name)
17: {
18:     int fd;
19:     struct stat st;
20:
21:     /* ファイルのサイズを調べる */
22:     if (stat (&st, file_name) < 0) {
23:         fprintf (stderr, "%s が見つかりません\n", file_name);
24:         exit (1);
25:     }
26:
27:     /* メモリを確保する。サイズは st.st_size */
28:     file_contents = (unsigned char *) malloc (st.st_size);
29:     if (file_contents == NULL) {
30:         fprintf (stderr, "メモリが足りません\n");
31:         exit (1);
32:     }
33:
34:     /* ファイルをオープンする */
35:     if ((fd = open (file_name, O_RDONLY | O_BINARY)) == NULL) {
36:         fprintf (stderr, "%s がオープンできません\n", file_name);
37:         exit (1);
38:     }
39:
40:     /* 読み込んで... */
41:     read (fd, file_contents, st.st_size);
42:
43:     /*クローズ */
44:     close (fd);
45: }
```

動的メモリ確保の場合，実行時にメモリが足りなくなるとmalloc関数がエラーを返すため「安心して使うことができない」のではないか，という不安感もあります。確かに配列は実行さえできれば，途中でメモリが足りなくなるというようなことはありません。その意味ではより安心かもしれませんが，結局は，本当に「メモリが足りない」のであれば，配列を用いたプログラムは実行できません。

### ◆ 動的メモリ確保の短所

動的メモリ確保の短所は，実は，かなりあります。それではその最も大きな短所(考え方にもよるかもしれません)を，2つあげてみましょう。

1. **自分でメモリ管理をしなければならない**

   配列はextern型，static型あるいはauto型にしろ，C言語で定義されている変数の**スコープ**にしたがって管理されますが，malloc関数などを用いて動的に確保したメモリ領域は自分で管理しなくてはなりません。メモリが必要になった時点で「自分で」確保しなければならないし，不要になった時点で「自分で」解放しなくてはなりません。たとえばList 2-15を見てください。この例では(A)の時点で(34行目)，もはや変数new_areaが指す領域は必要なくなっているので解放する必要があります。このことを忘れているため，このルーチンは呼び出されるたびにメモリのムダ使いをします。

**List 2-15 ● 動的メモリ確保の短所**

```
 1: /*
 2: ** free_err.c:
 3: **   次のルーチンでは確保したメモリ領域が不要になったときに解放する
 4: **   ことを忘れている。したがって，ルーチンが呼ばれるたびに無限に
 5: **   メモリを消費している
 6: */
 7: #include <stdlib.h>
 8: #include <stdio.h>
 9: #include <string.h>
10:
11: void concatCall (const char *string1, const char *string2)
12: {
13:     int length1, length2;
14:     char *new_area;
15:
16:     /* string1 と string2 の長さを求める */
17:     length1 = strlen (string1);
18:     length2 = strlen (string2);
19:
20:     /* 2 つの文字列を合わせたサイズを確保する */
21:     new_area = (char *) malloc (length1 + length2 + 1);
22:     if (new_area == NULL) {
23:         fprintf (stderr, "メモリが足りません\n");
24:         exit (1);
25:     }
26:
27:     /* 2 つの文字列をつなげる */
28:     strcpy (new_area, string1);
29:     strcat (new_area, string2);
30:
31:     /* つなげた文字列を表示する */
32:     puts (new_area);
33:
34:     /*
35:     ** (A) 本来ならば，ここで free (new_area) しなければなら
36:     ** ないが忘れている
37:     */
38: }
```

**2. 実行コストがかかる**

配列とは異なり，動的メモリ確保は必要になった時点で空いているメモリ領域を探してこなくてはなりません。そのため，実行時にそれなりの実行コスト[4)]がかかります。ほんの少しの回数ならば負担にはなりませんが，数百回，数千回，数万回実行されたら無視できなくなります[5)]。

4) ヒープ領域の管理アルゴリズムや状況によります。

5) *XC*では，メモリのフラグメンテーションが増大すると極端に遅くなります。

### ◆ 両者の選択

配列と動的メモリ確保には，それぞれに以上のような長所/短所が存在しています。メモリ領域(配列といってもよいでしょう)を必要とするすべての場所で，作成するプログラムの性格/用途などを考えて，どちらか適した方法を選択する必要があります。一般には，次のような判断基準が考えられます。

- **必要とする領域のサイズが固定ならば配列を利用する**
- **実行コストを抑え，高速に実行したいならば配列を利用する**

- **必要とする領域のサイズが可変あるいは予測できないならば，動的メモリ確保を使用する**
- **プログラムの実行に制限事項があってもよければ配列でもよい**
- **プログラムの実行に制限事項があってはいけないならば，動的メモリ確保を使用する**
- **必要とする領域そのものの必要数が可変あるいは予測できなければ，動的メモリ確保を使用する**

これらの判断基準は不確実ですから，実際に試してみて，考えるのがいちばんです。ところで，配列の長所も動的メモリ確保の長所も，そのどちらも合わせもった方法があります。alloca 関数は，これら 2 つの選択肢以外に加えてみてもよいかもしれません。詳しくは後述する「スタックからメモリを確保」(P.84)を参照してください。

**プログラムに制限をつけるのは好ましくない**

## ❖ ヒープ領域の使い方

以上で，ヒープ領域とそれを用いた動的メモリ確保について詳しく見てきました。本セクションでは，それらを実際に用いる方法について説明していきます。

### ◆ メモリブロックを確保する

ヒープ領域からメモリ領域を確保するためには，malloc 関数を使用します。malloc 関数は要求されたサイズのメモリ領域をヒープ領域から割り当て，その領域へのポインタを void *型で返します[6]。

*LIBC*ではこのポインタはすべてロングワードデータ境界，つまり MC68020 以上の MPU にとって最もきびしい**アラインメント**に調整されます。これは，C 言語で使用されるいかなるデータ型のポインタにでも**キャスト**することができ，しかも高速にアクセスできることを保証しなければならないからです。

6)古い C 言語では void * 型 の代わりに，char * 型 がよく用いられていました。

Fig. 2-4 を見てください。この図ではアドレス$200000 からバイトデータ，ワードデータ，ロングワードデータを並べたようすを図示しています。MC68000 ではバイトデータはどんな位置からも読み出すことができますが，ワードデータ/ロ

| | +0 | +1 | +2 | +3 |
|---|---|---|---|---|
| $200000 | BYTE | BYTE | BYTE | BYTE |
| $200004 | WORD | | WORD | |
| $200008 | LONG WORD | | | |

**Fig.2-4 ● MC68000 のアラインメント**

ングワードデータは偶数アドレスからしか読み出すことができません。この制限はMC68020以上のMPUではなくなりましたが，奇数アドレスからのワードデータ読み出しは，偶数アドレスからの読み出しに比べると非常に時間がかかるので，実際にはほとんど使われません。またMC68020以上では，同様に奇数ワードアドレスからのロングワードデータの読み出しに非常に時間がかかります。

またmalloc関数は，必要なメモリブロックが得られないときはNULLを返します。正しくメモリブロックが得られたかどうか，必ずチェックすることが必要です (List 2-16参照)。

**malloc関数の戻り値は必ずチェックすること**

**List 2-16 ● malloc関数の正しい用法**

```
 1:  /*
 2:  ** xmalloc.c:
 3:  **   次の関数 xmalloc はいろいろなプログラムでよく使われる。同様に,
 4:  **   xrealloc も用意するなど, エラーチェックつきのヒープ関数を作成
 5:  **   しておくと非常に便利である
 6:  */
 7:  #include <stdio.h>
 8:  #include <stdlib.h>
 9:
10:  void *xmalloc (size_t size)
11:  {
12:      void *ptr;
13:
14:      /* エラーチェックつき */
15:      ptr = (void *) malloc (size);
16:      if (ptr == NULL) {
17:          perror ("xmalloc");
18:          exit (1);
19:      }
20:
21:      /* ptr を返す */
22:      return ptr;
23:  }
```

そして非常に重要な注意点として，

**メモリブロックのサイズを越えてアクセスしてはならない**

ということがあります。たとえば，200バイト分として確保したメモリ領域に対して，200バイトを越えるデータを格納したりしないでください。まったく関係のないほかのデータが破壊されるばかりか，ヒープ領域の管理そのものが正常に実行できなくなる可能性があります。こうした場合，実際に問題となっている場所とは異なる場所に異常が現れるので，デバッグがやっかいになります。

たとえば，確保したメモリ領域のサイズは普通把握しているものですが，ふとしたことで，気づかないうちに (つまり想像していなかった状況で) このサイズを越えてしまうことがあるものです。どうしても納得のいかないようなデータ破壊があった場合，別の場所を疑ってみるのも手かもしれません。たとえば，List 2-17のようなことが起こっていませんか。

**List 2-17 ● ヒープ領域の破壊**

```
 1:  /*
 2:  ** heap_corup.c:
 3:  **   知らず知らずヒープ領域を破壊してしまう例
 4:  */
 5:  #include <stdlib.h>
 6:
 7:  /* 与えられた要素数 size の 2 つの配列の各要素を足し合わせる */
 8:  void addTwoArrays (int *array1, int *array2, int size)
 9:  {
10:      int i;
11:
12:      /*
13:      ** 次の i <= size は本当は i < size のまちがいである。この
14:      ** まちがいによって，配列の本当のサイズを 1 つオーバーして
15:      ** しまう。その結果，どこか別のデータが破壊される可能性が
16:      ** あるが，下のまちがいが原因だとはなかなか気づかない
17:      */
18:      for (i = 0; i <= size; i++)
19:          array1[i] = array1[i] * 3 + array2[i];
20:  }
```

### ◆ メモリブロックのサイズ変更

一度 malloc 関数によって確保されたメモリブロックのサイズを変更するときには，realloc 関数を使用します。ライブラリの実装方法にもよりますが，*LIBC* では，realloc 関数が指定されたメモリブロックのサイズを望みのサイズに変更するとき，次の 2 つの方法を使います。

1. **空きスペースを有効利用する**
   malloc 関数は，普通指定されたサイズよりも少しだけ余裕を見てメモリブロックを確保します。ですから，2 バイトのメモリ領域を確保したからといっても，実際に 2 バイトしかないとはかぎりません。事実，*LIBC* では，すべて 16 の倍数に調整されます。この性質を利用し，変更するサイズがこの余裕分でまかなえる範囲内ならば，管理上のサイズを変更するだけですみます。また変更したいメモリ領域の後ろに，空いているメモリ領域があれば，それを用いることもできます。
2. **別の場所に移動する**
   どうしても簡単に変更できない場合，realloc 関数は指定されたサイズに見あうメモリブロックを別の場所に確保し，現在のメモリ領域の内容をすべてコピーします。

(2.) の場合，メモリブロックの中身は同じでも位置が変わってしまいます。これまで使っていたメモリブロックはすでに存在しませんから，元の位置を指すようなポインタがあれば，新しい位置を指すように変更しなければなりません。

7)すでに説明したとおり，Human68k の空きメモリ容量は増えません。

### ◆ メモリブロックを解放する

必要なくなったメモリブロック[7]は，free 関数を用いて解放することができます。解放されたメモリはヒープ領域に空きメモリブロックとして戻され，後で malloc 関数や realloc 関数によって再利用されます。

いったん解放され，アプリケーションの手から離れたメモリブロックは内容が保証されません。解放された後のメモリブロックの中身にアクセスするようなプログラム (たとえば List 2-18 のようなプログラム) は，書かないようにしてください。

**List 2-18 ● free 関数の誤った使い方**

```
 1: /*
 2: ** list_free.c:
 3: **   線形リストに組み合わされたメモリブロックを解放する
 4: */
 5: struct pList {
 6:     struct pList *next; /* 線形リストの次のブロックを指す */
 7:     double x, y, z;     /* 3 次元座標 */
 8: };
 9:
10: /* まちがい */
11:
12: void freePointList1 (struct pList *top)
13: {
14:     /* 線形リストを先頭から順番に解放 */
15:     for (; top; top = top->next)
16:         free (top);
17:
18:     /*
19:     ** top = top->next の部分は free (top) で top が指すメ
20:     ** モリブロックが解放された後で実行される。解放された
21:     ** メモリブロックである top をアクセスしてはならない
22:     */
23: }
24:
25: /* 正しくは ... */
26:
27: void freePointList2 (struct pList *top)
28: {
29:     struct pList *next;
30:
31:     /* 線形リストを先頭から順番に解放 */
32:     for (; top; top = next) {
33:         next = top->next;
34:         free (top);
35:     }
36: }
```

# 2.5 ………………………………… スタック領域

## ❖ スタック領域の使われ方

GCC などを含め，一般の C 言語ではどのようにスタック領域を利用しているのでしょうか。スタック領域の詳しい説明にはいる前に，まず簡単におさらいしておきます。

### ◆ 引数の渡し方

GCC[1]では，引数をスタックと呼ばれる領域に格納して，関数の呼び出しを行っています。細かいことを説明してもわかりにくいでしょうから，具体的に順を追って説明していきましょう。たとえば，char 型と char *型の 2 つの引数をもつ関数 foo を呼び出すことを考えます。foo 関数のプロトタイプは次のようになっています。

1)C というよりは，C コンパイラによって生成された実行ファイルです。

**List 2-19 ● foo 関数のプロトタイプ**

```
1:  extern void foo (char letter, char *message);
2:
3:  void fatalError (void)
4:  {
5:      /* foo を呼び出す */
6:      foo ('A', "Error message");
7:  }
```

Fig. 2-5 を見てください。まず，呼び出し元の関数は引数を後ろからスタック上に格納していきます。最初は *letter*，次に *error*，そして最後に foo 関数から戻るときの戻り先アドレスです。ここで注意することが 1 つあります。1 バイトのデータである char 型の引数が，4 バイトの領域を占めているということです。

引数は，構造を簡単にするためやアラインメント[2]や効率化のために，たとえ 1 バイトのデータでも，ある程度の領域を使って格納されているのが普通です。この最低バイト数は MPU の特性によるわけですが，**X68000** で使われている MC68000 や **X68030** で使われている MC68030 では 4 バイトデータ，つまり 1 ロングワードデータが最適とされています。

2) MC68000 では，奇数アドレスからのワードデータ/ロングワードデータ読み出しはできません。

**char 型も short 型も 1 ロングワードデータで扱われる**

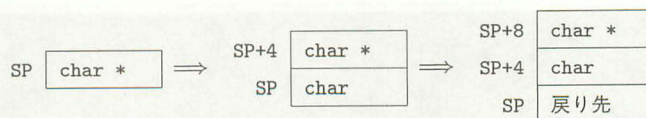



Fig.2-5● スタックの動き

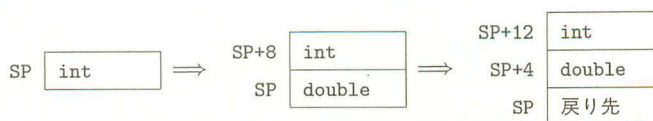



Fig.2-6● double 型は8バイトを占める

呼び出されたfoo関数は，どこを見れば目的の引数を取り出せるかを知っています。たとえばこの場合，*error*が現在のスタックポインタの位置から計算して+4バイトの位置から1ロングワードデータ，また*message*が+8から1ロングワードデータに納められていることがわかっています。このようにして，関数と関数との間で，引数のやりとりが成立するわけです。

ではfoo関数の処理が終わり，もはや引数が必要なくなるとどうなるでしょうか。foo関数はスタック上に記憶されている戻りアドレスへジャンプし，呼び出し元の関数へ戻ってきます。そして，呼び出し元の関数がスタックを調整して引数を捨てます[3]。

3)引数を捨てるとは，つまりスタックポインタの値に引数が占めていたバイト数を加算して，元に戻すということです。

ちなみに4バイトを越えるようなデータを引数に渡すと，どうなるでしょうか。答えは「そのままスタックに格納する」です。引数の占める最低のバイト数は存在しますが，事実上，最大バイト数は存在しません。もし，256バイトにもなるような大きな構造体を**ポインタ渡し**(Call by reference)ではなく，**値渡し**(Call by value)で渡すようにすれば，当然256バイトがまるごとスタック上に格納されることになります(Fig. 2-6参照)。

### ◆ スタックからメモリを確保

このように関数の呼び出しにはスタックが用いられますが，このスタックからmalloc関数のように，任意のサイズのメモリを確保することもできます。そのためには，alloca関数を使います。この関数は，ちょうどauto型の変数と同じ感覚でメモリ領域を確保することができます。

ですから，alloca関数で確保されたメモリ領域は解放する必要がありません。このメモリ領域はスタック上にあるために，ほかのauto型変数と同じように，現在のスコープから出ると自動的に解放されます[4]。

4)実際には，スタックポインタの値を調整しているにすぎません。

**alloca関数で確保した領域は自動的に解放される**

GCCでは可変長のauto型配列を宣言できる機能が拡張されていますが，これは，まさにこのalloca関数の機能と同じものです。List 2-20を可変長配列を使って書き直すと，List 2-21のようになります。

**List 2-20 ● alloca 関数の使い方**

```
 1: /*
 2: ** allocatest.c:
 3: **    2 つの引数を連結して表示する
 4: */
 5: #include <alloca.h>
 6: #include <stdio.h>
 7:
 8: void printConcat (const char *string1, const char *string2)
 9: {
10:     int size;
11:     char *result;
12:
13:     /* 文字列の長さを計算 */
14:     size = strlen (string1) + strlen (string2) + 1;
15:
16:     /* その長さの領域を確保 */
17:     result = (char *) alloca (size);
18:
19:     /* 文字列を連結 */
20:     strcpy (result, string1);
21:     strcat (result, string2);
22:
23:     /* 結果を表示する */
24:     puts (result);
25:
26:     /* この関数を抜けると result のバッファは解放される */
27: }
```

**List 2-21 ● 可変長配列の使い方**

```
 1: /*
 2: ** vararray.c:
 3: **    2 つの引数を連結して表示する
 4: */
 5: #include <stdio.h>
 6:
 7: void printConcat (const char *string1, const char *string2)
 8: {
 9:     /* 文字列の長さを計算 */
10:     char result[strlen (string1) + strlen (string2) + 1];
11:
12:     /* 文字列を連結 */
13:     strcpy (result, string1);
14:     strcat (result, string2);
15:
16:     /* 結果を表示する */
17:     puts (result);
18: }
```

### ◆ スタックオーバーフロー

**スタック領域**は**ヒープ領域**とは異なり，足りなくなったからといって自動的に大きさが拡張されたりはしません[5]。プログラムを起動したときに割り当てられた，一定サイズの領域を使い回すことしかできないのです。もしも，この領域をすべて使い切ってしまったらどうなるでしょうか。答えは**「スタックオーバーフロー」**です。このスタックオーバーフローという現象は，概念としては非常に単純明解なのですが，実際に発生したときには非常に見分けにくいものです。

5)ヒープ領域は足りなくなると，自動的に大きさが拡張されていきます。

スタックオーバーフローが起こると，これまで説明したような引数や戻りアドレスがスタック領域をはみ出して格納されていきます。Fig. 2-7を見てください。***LIBC***では起動時に，この図のようにスタック領域を割り付けて使うようにしています。スタックオーバーフローが発生すると，結果としてスタック領域のすぐ隣りのデータ領域が破壊され，プログラム内部のグローバル変数などの値が予期しない値になってしまいます。さらに破壊が進めば，データ領域どころかプログラム領域までが破壊され，プログラムはもはや実行できなくなってしまうでしょう。

しかしプログラムが異常終了したり，暴走したりしたからといって，それがスタックオーバーフローによって起こされたものなのかどうかは，一概にはわかりません。単なるプログラムのバグなのかもしれません。また逆にいえば，スタックオーバーフローによって引き起こされたことなのに，それをバグだと思って，一所懸命にソースコードとにらめっこするはめになるかもしれません。

#### 原因はスタックオーバーフローか，それともバグか

この原因を調べるため，GCCでは関数ごとにスタックオーバーフローをチェックするルーチンを付加して，**スタックチェック**を行うようにすることができます。このスタックチェックルーチンは，関数が呼び出されるごとに今のスタックポインタの位置を調べ，それがスタック領域からはみ出していればエラーとして表示するものです。***LIBC***もこの機能をサポートしており，スタックオーバーフローが発生すると，次のようなメッセージを表示して異常終了します。

```
libc: stack over flow
```

| メモリ管理ブロック |
|---|
| プロセス管理ブロック |
| プログラム領域 |
| データ領域（初期値なし） |
| データ領域（初期値あり） |
| スタック領域 |
| ヒープ領域 |

**Fig.2-7● *LIBC*のメモリ構成**

確かに余計な処理が関数ごとに加えられるわけですから，サイズも少々大きくなりますし，実行速度も犠牲になります。しかし，それらよりもスタックがオーバーフローしないかどうかをチェックするほうが大切な場合には，この機能を活用してみてください。特に，デバッグ時に有効な機能ではないでしょうか。使用するには，次のように指定します(buggy.c というファイル名とします)。

```
A:\> gcc -fstack-check buggy.c
```

### ◆ スタックサイズの指定

このようなスタックオーバーフローを引き起こさないためにも，プログラムにはそれに適したサイズのスタック領域を与えてやる必要があります。*LIBC*はデフォルトで32Kバイトのスタック領域をプログラムに割り当てていますが，もしもこのサイズを変更したいならば，次のような方法でプログラムを作成してください。

**○ つねに指定したサイズのスタック領域を割り当てたい場合**

*LIBC*は起動時に変数_stacksizの値を見て，スタック領域の大きさを決定しています。ですから，自分のプログラム内でこの変数を再定義してやればよいのです。たとえば，プログラムに128Kバイト(131072バイト)のスタック領域を与えたいときは次のようにします。

**List 2-22 ● スタックサイズを指定する**

```
 1:  /*
 2:  ** hello.c:
 3:  **    このプログラムはスタックに 128K バイト必要とする
 4:  */
 5:  #include <stdio.h>
 6:
 7:  int _stacksiz = 128 * 1024; /* バイト単位で指定すること */
 8:
 9:  int main (void)
10:  {
11:      printf ("Hello world !!\n");
12:      return 0;
13:  }
```

**○ 起動時に一時的にスタックサイズを変更したい場合**

*LIBC*はコマンドラインから特殊なオプション“-+-s:”を用いて，スタック領域のサイズを変更することができます。たとえば，*LIBC*を使って作成したプログラム“hello”のスタックサイズを，一時的に128Kバイトにして起動したい場合は次のようにします。オプションの次の数字は，スタック領域のバイト単位のサイズを表しています。なお，指定した“-+-s:”オプションは*LIBC*が取り込んでしまうので，プログラムには渡されません。

```
A:\> hello -+-s:131072
```

## ❖ 可変長引数

**可変長引数**とは，数の決まっていない引数列のことをいいます。たとえばprintf関数などは引数の数が決まっておらず，必要に応じて増えたり減ったりします。このような，あらかじめ引数の数やそのデータ型などが不明な場合は，どのように引数を取り扱えばよいのでしょうか。本セクションでは，これらの扱い方について触れることにしましょう。

### ◆ 可変長引数とは

可変長引数を取り扱うインタフェイスとして，*LIBC*は2つのインタフェイス，<stdarg.h>と<varargs.h>をもっています。しかし<varargs.h>は*K&R*時代のC言語で使われていた方法であり，いささか古くさい方法ですので[6]，ここでは解説しません。その代わりに，*ANSI C*で規定された<stdarg.h>インタフェイスのほうを使うことにします。どちらのインタフェイスも基本的には同じものですから，暇があれば，<varargs.h>のほうも自分で解析してみるのもよいかもしれません。

6)実現方法が*K&R*の関数宣言の方法に依存しており，*ANSI C*時代にはそぐわないものです。

### ◆ 準備

可変長引数を扱うには，まず**可変長引数リスト**を求めなくてはなりません。可変長引数リストとは，スタック上の引数を順番に取り出していくためのポインタと考えてよいでしょう。事実，そのとおりです。わかりやすいように，再び図を使って考えてみましょう。一例として，printf関数を取り上げます。

**List 2-23 ● 可変長引数をもつ関数**

```
 1:  /*
 2:  ** printf.c
 3:  **    可変長引数をとる関数を呼び出す
 4:  */
 5:  extern int printf (const char *format, ...);
 6:
 7:  void foo (void)
 8:  {
 9:      printf ("I am %s, and %d year(s) old.\n", "Michael", 10);
10:  }
```

Fig. 2-8を見てください。可変長の引数を取り出すには，まず$SP+8$の位置を求めなければならないことがおわかりでしょうか。この位置から後ろの部分を**可変長引数リスト**と呼びます。<stdarg.h>インタフェイスでは，これを求め

**Fig.2-8**● 可変長引数をもつ関数の呼び出しスタック

るために va_start マクロを使います。va_start マクロは通常の引数リストの最後の引数の位置を求め[7]，そこから可変長引数リストの先頭アドレスを求めます(List 2-24 参照)。ですから可変長引数を扱う関数は，最低でも 1 つの通常引数をもたなくてはなりません。これが唯一の制限事項です。

7)これは &演算子を使えば，&format で計算できます。

**最低 1 つは普通の引数が必要である**

**List 2-24**● va_start を使って引数リストの先頭を求める

```
 1:  /*
 2:  ** vastart.c:
 3:  **    可変長引数リストの先頭アドレスを求める
 4:  */
 5:  #include <stdarg.h>
 6:
 7:  int printf (const char *format, ...)
 8:  {
 9:      va_list ap; /* 可変長引数リストを扱うポインタ型 */
10:
11:      /* ap にアドレスが格納される */
12:      va_start (ap, format);
13:
14:          :
15:          :
16:
17:      /* リストへのアクセスが終わるときには va_end */
18:      va_end (ap);
19:  }
```

### ◆ 引数を取り出す

可変長引数リストが求められたところで，いよいよ引数を取り出すことにしましょう。リストから必要な引数を取り出すには，va_arg マクロを使います。このマクロに，引数リストへのポインタと取り出したい引数の型情報を与えてやります。先ほどの例に戻って考えてみましょう。たとえば，char *型の引数 “Michael” と int 型の引数 10 を取り出すには，List 2-25 のようにプログラミングします。

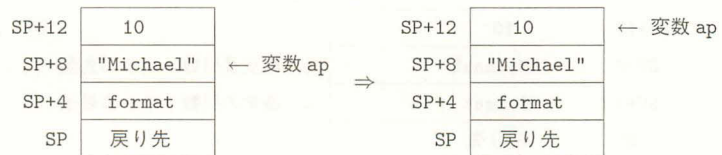



**Fig.2-9** ● va_arg によるポインタの変化

**List 2-25** ● va_arg マクロの使い方

```
 1:  /*
 2:  ** vaarg.c:
 3:  **    可変長引数リストからデータを取り出す
 4:  */
 5:  #include <stdarg.h>
 6:
 7:  int printf (const char *format, ...)
 8:  {
 9:      char *arg1;
10:      int arg2;
11:      va_list ap;
12:
13:      va_start (ap, format);       /* ap を初期化 */
14:      arg1 = va_arg (ap, char *); /* ap から char * 型の引数を取り出す */
15:      arg2 = va_arg (ap, int);    /* ap から int 型の引数を取り出す */
16:      va_end (ap);                /* 終わり */
17:  }
```

List 2-25 中では，va_arg マクロを 2 回呼び出していますが，va_arg マクロは呼び出されるたびに変数 ap を更新しています (Fig. 2-9 参照)。

このとき，va_arg マクロに指定する型情報をまちがえないように気をつけてください。もし誤った型情報を指定すると，その引数が正しい値として取り出せないばかりかポインタが狂ってしまい，それ以降の引数がすべて無効になってしまうことがあります[8)]。可変長引数の部分は，*ANSI C*のプロトタイプチェックが機能しませんから，自分で注意しなければなりません[9)]。

8)たとえば，double 型の引数を float 型で取り出したりしないでください。

9)printf 関数などの可変長引数の型チェックを行う機能が GCC ver.2 に追加されましたが，これは GCC が関数の仕様を知っているからにすぎません。

### スタックに格納した型と取り出すときの型は同じでなければならない

#### ◆ 可変長引数の実際

最後に，ここまで説明してきた知識をベースに printf 関数に似た簡単なフォーマット表示関数を作成してみました (List 2-26 参照)。後は皆さんで考えてみてください。

**List 2-26** ● 可変長引数の実際例

```
1:  /*
2:  ** vatest.c:
3:  **    可変長引数を使ったフォーマット表示
4:  */
5:  #include <stdarg.h>
6:  #include <stdio.h>
```

```
 7:
 8: void formatOut (const char *format, ...)
 9: {
10:     char ch;
11:     va_list ap;
12:
13:     /* ap を初期化 */
14:     va_start (ap, format);
15:
16:     /* format 文字列を調べていく */
17:     while (ch = *format++) {
18:
19:         /* % 以外の文字はそのまま出力する */
20:         if (ch != '%')
21:             fputc (ch, stdout);
22:
23:         /* % ならばそれに続く文字によって分類 */
24:         else
25:             switch (*format++) {
26:             case 's':
27:                 fputs (va_arg (ap, char *), stdout);
28:                 break;
29:             case 'c':
30:                 fputc (va_arg (ap, int), stdout);
31:                 break;
32:             default:
33:                 fputc ('%', stdout);
34:                 break;
35:             }
36:     }
37:
38:     /* ap によるアクセスを終わる */
39:     va_end (ap);
40: }
41:
42: int main (void)
43: {
44:     formatOut ("%s is %c\n", "First char", 'F');
45: }
```

このプログラムをコンパイルし実行させると，画面には次のように表示されます。

```
A:\> gcc -O vatest.c
A:\> vatest.x
First char is F
A:\>
```

# 2.6 …………………………………… シグナル機構

## ❖ シグナル機構について

シグナルとは，ハードウェアおよびソフトウェアからランダムに発生する割り込みのことです。C言語ではこれらのシグナル割り込みを処理するために，**シグナルハンドラ**を記述することができます。本セクションでは，シグナルの取り扱いについて解説します。

### ◆ Human68k とシグナル

**Human68k** というよりは *XC* では，基本的に CTRL + C の押下げによって発生する SIGINT というシグナルしか扱えません。しかしシグナルの実体は割り込みですから，それ以外のシグナルであっても **X68000** および **X68030** の各種割り込みを，ライブラリが直接扱えば操作することが可能になります。

確かに，本来シグナル機構は OS がサポートすべき範囲のものなので，それをユーザレベルの *LIBC* が処理するというのは，あまりよいとはいえないでしょう。しかし *LIBC* では「やれることはやってみる」というスタンスから，従来はあまり重視されなかったシグナル機能について，できる範囲内でサポートすることにしました。

### ◆ シグナルの種類

まず，実際にどのようなシグナルが発生するのかについて確認しておきましょう。*LIBC* が処理するシグナルには，Table 2-18 のようなものがありますが，これらは大別すると，次の3つに分類されます。

○ **発生区分がソフトウェア例外のもの**

シグナル SIGABRT，SIGKILL，SIGTERM，SIGSTOP，SIGUSR1，SIGUSR2 はソフトウェア例外による割り込みです。ソフトウェア例外というのは *LIBC* が自分で発生させる trap 14 の例外のことで，おもにほかの処理系との互換性のために用意されています。

○ **発生区分が Human68k 例外のもの**

シグナル SIGINT は，**Human68k** 例外による割り込みです。**Human68k** 例外というのは，**Human68k** がキーボードをチェックして発生させる trap

**Table 2-18 ● *LIBC*のシグナル一覧**

| シグナル | 発生区分 | 内容 |
|---|---|---|
| SIGABRT | ソフトウェア例外 | ABORT シグナル |
| SIGINT | **Human68k** 例外 | CTRL+C インタラプト |
| SIGILL | ハードウェア例外 | 不正な命令を実行した |
| SIGFPE | ハードウェア例外 | 0 による除算 |
| SIGKILL | ソフトウェア例外 | KILL シグナル |
| SIGBUS | ハードウェア例外 | アドレスエラー |
| SIGSEGV | ハードウェア例外 | バスエラー |
| SIGALRM | ハードウェア例外 | リアルタイムクロック割り込み |
| SIGTERM | ソフトウェア例外 | 終了シグナル |
| SIGEMT | ハードウェア例外 | 未定義トラップ |
| SIGSTOP | ソフトウェア例外 | STOP シグナル |
| SIGUSR1 | ソフトウェア例外 | USR1 シグナル |
| SIGUSR2 | ソフトウェア例外 | USR2 シグナル |

13 の例外のこと，ソフトウェア例外の一種になります。ただし *LIBC*では，`trap 13` は利用していません。このことについての詳細は後で述べます。

○ **発生区分がハードウェア例外のもの**

シグナル SIGILL，SIGFPE，SIGBUS，SIGSEGV，SIGALRM，SIGEMT はハードウェア例外による割り込みです。ただし正確には，ハードウェア例外を **Human68k** が処理した後に発生する，ソフトウェア例外のことを意味します[1]。SIGFPE と SIGALRM を除くこれらのシグナルは，現在実行中のプロセスが回復不能な状況になったことを通知するためのものです。

1)ただし，SIGALRM は純粋にハードウェア例外です。

### ◆ シグナルハンドラの設定

Table 2-18 で示したシグナルは，それぞれ内容で述べたような状況が発生すると，現在実行中のプログラムに割り込みとして通知されます。実行中のプログラムは一時中断され，ライブラリによってそれぞれのシグナルに対応した**シグナルハンドラ**が呼び出されます。

ユーザがシグナルハンドラに何も設定しなくても，ライブラリにはそれぞれのシグナルに対応した標準のシグナルハンドラが存在しています。これら標準のシグナルハンドラは，Table 2-19 のような動作をします。

特に複雑な処理をしないプログラムならば，標準のシグナルハンドラのまま利用しても問題ありません。しかし，ある程度エラー処理をしなければならないプログラムを作成するならば，ユーザが独自にシグナルハンドラを用意する必要があるでしょう。独自のシグナルハンドラを設定するには，`signal` 関数を用います。

### ◆ シグナルハンドラの作り方

シグナルハンドラは，`int` 型の引数 (シグナル番号) を 1 つだけもつ[2]`void` 型の関数です。割り込み処理といっても，実際のめんどうな処理はすべてライブラリ

2)UNIX などでは，割り込みに関する情報などがより詳しく渡されます。

Table 2-19 ● 標準のシグナルハンドラ

| シグナル | 処 理 |
|---|---|
| SIGABRT | 強制終了 |
| SIGINT | 強制終了 |
| SIGILL | 強制終了し，メモリイメージをファイルにダンプ |
| SIGFPE | 強制終了し，メモリイメージをファイルにダンプ |
| SIGKILL | 強制終了 |
| SIGBUS | 強制終了し，メモリイメージをファイルにダンプ |
| SIGSEGV | 強制終了し，メモリイメージをファイルにダンプ |
| SIGALRM | 強制終了 |
| SIGTERM | 強制終了 |
| SIGEMT | 強制終了し，メモリイメージをファイルにダンプ |
| SIGSTOP | 無視 |
| SIGUSR1 | 強制終了 |
| SIGUSR2 | 強制終了 |

が行っていますから，シグナルハンドラではいくつかの注意点を守るだけでかまいません。

List 2-27を見てください。これはSIGSEGVシグナルに対するシグナルハンドラです。バスエラーによって回復不能になり，SIGSEGVシグナルが発生すると，メッセージを表示して終了するように作成されています。

**List 2-27 ● SIGSEGV シグナルハンドラ**

```
 1:  /*
 2:  ** sigsegv.c:
 3:  **   SIGSEGV が発生すると，メッセージを表示して終了させるための
 4:  **   シグナルハンドラ
 5:  */
 6:  #include <signal.h>
 7:  #include <sys/dos.h>
 8:
 9:  void sigsegv_handler (int sig_number)
10:  {
11:      _dos_print ("バスエラーが発生しました..... 終了します。\r\n");
12:      exit (1);
13:  }
```

### ◆ シグナルハンドラの注意点

シグナルハンドラは普通の関数と同じように書くことができると述べましたが，それでも本質は割り込み処理なので，いくつかの制限があります。また，行ってはいけない禁止事項もあります。

#### ○ ハードウェア例外によるシグナルは復帰させてはならない

SIGFPEとSIGALRMを除くハードウェア例外シグナルは，プログラムが回復不能な状況になったことを表すシグナルです。これらのシグナルハンドラでは，必ず最後にプログラムを終了させてください (List 2-27参照)。もし終了

させなかった場合は，正常に動作しなくなるだけでなく，**Human68k**のシステムそのものが回復不能になることがあります。

○ **リエントラントでない関数を使用してはならない**

シグナルハンドラは割り込み処理です。現在実行中の処理が任意の場所で中断したままになっています。ですから，これらの中断している処理に副作用をおよぼすような処理をしてはいけません。つまり，リエントラント(再入可能)な関数しか使用することができません。

これはハードウェア例外シグナルに対するシグナルハンドラなど，シグナルハンドラ内部でプログラムを終了してしまう場合でも同じことです。List 2-27で`_dos_print`関数というDOSコールを使用していて，`printf`関数や`puts`関数などの`stdio`ライブラリを用いていないのはこのためです[3]。

リエントラントな関数とは**副作用**のない関数のことで，`auto`型変数だけを利用し，`static`型や`extern`型の変数を参照/変更しない関数のことをいいます。また，当然どこかのメモリ領域を参照したり，書き換えるようなこともしてはいけません。外部の状況に左右されないことが，リエントラントの条件だからです。たとえば，List 2-28にリエントラントな関数とそうでない関数の例をあげてみました。

3)とはいえ，本当のところ，DOSコールがリエントラントとは誰も保証していませんから，これも誤りです。

**List 2-28 ● リエントラントな関数**

```
 1:  /*
 2:  ** reent.c:
 3:  **    リエントラントな関数，そうでない関数とは...
 4:  */
 5:
 6:  /* リエントラントな関数 */
 7:  void SimpleSample1 (double a, double b)
 8:  {
 9:      /* 引数だけで，外部と一切関係をもたない */
10:      return 2 * a * a + b;
11:  }
12:
13:  /* リエントラントでない関数 */
14:  void SimpleSample2 (int *array)
15:  {
16:      /* 外部のメモリ領域 array を参照している */
17:      return array[0] + array[1] + 10;
18:  }
```

*POSIX.1*などではリエントラントな関数とそうでない関数を明確に定義していますが，*LIBC*では明確にしていません。これからも仕様の変更などが十分に予想されるからです。ただし，我々がリエントラントな関数として作成した関数には`__const`型という関数属性の修飾子がついていますから，インクルードファイルを見れば，ある程度[4]リエントラントかどうか，調べることができます。実際に使ってみて試すという方法もありますが，副作用がどういう影響をおよぼすかについては，ある意味で確率的な問題ですから，あまりよい方法とはいえないでしょう。

4)あまり信用はしないでください。

いささか矛盾しますが，リエントラントでない関数を使うことが，必ずしもいけないわけではありません。というのも，発生するであろう副作用と現在実行中のプログラムの相関関係を正しく把握し，干渉が起こらないことを確認したうえでならば，リエントラントでない関数を利用することも可能だからです。

**中断したプログラムを邪魔してはならない**

### ◆ 理想的なシグナルハンドラ

こうしたことを考慮すると，理想的なシグナルハンドラとはごくごく単純な形に行きつくことがわかります。そして，事実そのような単純なシグナルハンドラこそが，最も推奨されるシグナルハンドラの書き方です (List 2-29 参照)。

**List 2-29 ● 理想的なシグナルハンドラの姿**

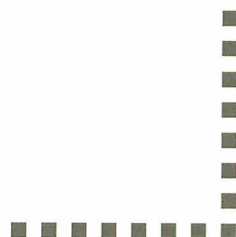
```
 1:  /*
 2:  ** sighand.c:
 3:  **    理想的なシグナルハンドラは割り込みフラグを立てるだけで
 4:  **    ある。外部変数を変更するため，リエントラントとはいえな
 5:  **    いが，割り込みのためだけに利用されるフラグなので，中断
 6:  **    されたプログラムとの副作用の相関関係はない
 7:  */
 8:  #include <signal.h>
 9:
10:  sig_atomic_t interrupt_flag = 0; /* 割り込みフラグ */
11:
12:  void sigalarm_handler (int sig_number)
13:  {
14:      interrupt_flag = 1;
15:  }
```

List 2-29 のプログラムでは sig_atomic_t 型の変数が用いられていますが，この sig_atomic_t 型という変数型は，シグナルハンドラで利用される割り込みフラグのためのデータ型のことです。*LIBC*ではこのデータ型を volatile int 型に定義していますから，もし int 型以外のデータ型をフラグに用いたい場合は，必要なデータ型に volatile 型の修飾子をつけてください。

この処理は，おもにコンパイラの最適化によって誤動作するのを防ぐためのものです。通常，コンパイラは処理がシーケンシャルに連続して行われるものとして最適化[5]を行います。ですから割り込み処理によって，値が変化する可能性があることを，教えてやらねばなりません。

5)要するに，フラグ変数の値がレジスタに割り当てられてしまうと，割り込みによって値が変化したことがわからないからです。

**割り込みフラグは volatile 型にすること**

Chapter 3

# Appendix A

Appendix A として，*LIBC*の用語解説と *LIBC*に関するややハイレベルな情報を掲載します。本章に記載した内容は，特定の *LIBC*バージョンに依存することがあり，将来は変更される可能性がありますから注意してください。

# A............................ *LIBC*の起動オプション

*LIBC*を使用して作成したプログラムを起動すると，起動時に指定した引数がそのまま引数の配列として`main`関数に渡されます。アプリケーションはこの引数の配列を調べて，オプションやファイル名などを取り出すわけですが，これと同様に*LIBC*への**起動オプション**というものが存在します。

*LIBC*のオプションとは，起動時に*LIBC*が解釈し，ライブラリ自体の動作を決定するためのものです。このオプションは*LIBC*に対するものなので，ライブラリが引数配列から抜き取って，`main`関数には渡されません。そのためこのオプションは，アプリケーションのオプションと区別するために(またアプリケーションの自由度を制限しないため)やや冗長な形式となっています。

現在，*LIBC*は次の**ライブラリオプション**(起動オプション)を解釈します。

○ `-+-s:`*bytes*

*bytes*で指定したバイト数をスタック領域に割り当てます。このオプションを指定しない場合，スタック領域は標準で32Kバイト確保されます。`-+-s:`と*bytes*の間は空白を入れず，また*bytes*はバイト単位で指定してください。

○ `-+-h:`*bytes*

*bytes*で指定したバイト数をヒープ領域に割り当てます。ヒープ領域は足りなくなった時点で自動的に拡張されるので，この値は初期サイズを指定するためのものです。このオプションを指定しない場合，ヒープ領域は標準で64Kバイト確保されます。`-+-h:`と*bytes*の間は空白を入れず，また*bytes*はバイト単位で指定してください。

○ `-+-p`

このオプションを指定すると，`main`関数を実行する前にアプリケーションを**スーパーバイザモード**に変更します。その結果，`I/O`コプロセッサを利用した数値演算が高速化されますが，当然，メモリ保護が効かなくなるなどの危険が生じます。このオプションを指定しない場合，アプリケーションは**ユーザモード**で動作します。

○ `-+-f`

*LIBC*は「*LIBC*の数学関数の特徴」(P.55)で解説したとおり，コプロセッサが利用可能ならば，それらを用いて数値計算を行います。ただし，いろいろな理由からコプロセッサの代わりにソフトウェアで計算したい場合があります。そのような場合は，このオプションを指定してプログラムを起動してください。*LIBC*は，数値演算を`FLOAT`パッケージを使用して行うようになります。

○ -+-g

C++プログラムを開発する場合,「libcplus.a」(P.32) で解説したとおり,“libcplus.a”をリンクする必要があります。しかし,もしこのライブラリをリンクし忘れたり,意図的にリンクしたくない場合もあるでしょう。そのようなときは,このオプションを指定することでプログラム中の**グローバルコンストラクタ/グローバルデストラクタ**を強制的に起動することができます。

たとえば,“compress.x”というある任意のプログラムを例に,ライブラリオプションの使用法を説明します。このプログラムは非常に多くのスタックを消費するので,あらかじめ多めにスタック領域を与えてやらねばなりません。“compress.x”を,256Kバイトのスタック領域とともに起動するには次のようにタイプしてください。画面の表示例中で,262144というのが256Kバイトをバイト単位で表現した値です。

```
A:\> compress.x bigfile -+-s:262144
```

この例ではcompress.xに,2つの引数bigfileと-+-s:262144が渡されていますが,このうち実際にmain関数に渡されるのはbigfileだけです。また,*LIBC*のライブラリオプションは任意の位置で指定することができます。

なお,ライブラリオプションの処理を行う関数についてはVol.2「Programmer's Reference」の_enargv関数に関する記述を参照してください。

## B………………………… *LIBC*のエラーメッセージ

どうしても必要な場合，*LIBC*は次のようなエラーメッセージを表示します。このエラーメッセージはプログラムの動作状況によらず，つねにコンソールに表示されます。

○ **スタックオーバーフロー**

下図は，**スタックオーバーフロー**が起こったことを知らせるメッセージです。*LIBC*はこのメッセージを表示した後，プログラムを強制終了させます。

GCCで-fstack-checkオプションを指定してコンパイルすると，スタックチェックつきの実行ファイルができあがります。この実行ファイルは関数呼び出しのたびに，**スタックポインタ**がスタック領域からはみ出していないかどうかをチェックします。もしスタック領域をはみ出し，暴走する可能性がある場合はこのメッセージが表示されます。

```
libc: stack overflow.
```

○ **メモリブロック配置エラー**

下図は，プログラムを実行するだけのメモリが足りない場合に表示されるメッセージです。*LIBC*はこのメッセージを表示した後，プログラムを強制終了させます。

このメッセージは，プログラムの実行に必要なスタック領域とヒープ領域が確保できないことを意味していますが，ヒープ領域の拡張とは関係がありません。main関数の処理が始まる以前に，スタック領域/ヒープ領域の配置を決定する段階で発生するエラーです。この段階では，アプリケーションは何も実行されていません。

```
libc: setblock failed.
```

# C………………………………… *LIBC*のエラーコード

*LIBC*で提供されている関数の多くは，実行に失敗した場合エラーを示す値を返すとともに，変数 errno にエラーの原因を示すコードを設定します。変数 errno およびエラーコードは<errno.h>に定義されており，次のような意味があります。

なお，文末に [II] とマークされているのは「*X680x0 Develop. & libc II*」に付属する *LIBC*で新たに使用されるようになったコードです。

- E2BIG　子プロセスを起動する際に，親プロセスから子プロセスに渡される引数が長すぎる
- EACCES　指定したパス名にアクセスできない [II]
- EAGAIN　リソースに一時的にアクセスできない [II]
- EBADF　使用していないか，ファイルハンドルか，あるいは不正なファイルハンドルを指定した
- EBUSY　リソースが使用中である [II]
- ECHILD　子プロセスが存在しない [II]
- EDEADLK　デッドロックが起きてしまう [II]
- EDEVFS　デバイスは指定することができない
- EDOM　数学関数で，関数の定義域外の引数を指定した
- EEXIST　同名のファイルがすでに存在している
- EFAULT　不正なアドレスを指定した
- EFBIG　ファイルが大きすぎる [II]
- EINTR　シグナルの割り込みによって処理が中断された
- EINVAL　不正な引数を指定した
- EIO　物理的なデータ入出力中に何らかのエラーが発生した
- EISDIR　ディレクトリを指定することはできない
- ELOOP　シンボリックリンクのネストが深すぎるか，どこかでループしている
- EMFILE　これ以上ファイルをオープンすることはできない
- EMLINK　これ以上ファイルをリンクできない [II]
- ENAMETOOLONG　ファイル名が長すぎる
- ENFILE　これ以上ファイルをオープンできない (システム全体) [II]
- ENODEV　指定したデバイスが見つからない [II]
- ENOENT　指定したファイル，あるいはディレクトリが見つからない

- ENOEXEC　　不正な実行フォーマットである [II]
- ENOLCK　　これ以上ロックできない [II]
- ENOMEM　　メモリが足りなくなった
- ENOSPC　　ディスクが一杯になったのでこれ以上書き込めない
- ENOSYS　　この機能は使用できない
- ENOTBLK　　指定したデバイスはブロックデバイスではない [II]
- ENOTDIR　　ディレクトリを指定しなければならない
- ENOTEMPTY　　ディレクトリが空でないので削除することができない
- ENOTTY　　キャラクタデバイスを指定しなければならない
- ENXIO　　指定したデバイス，アドレスが見つからない [II]
- EPERM　　禁止された操作を行おうとした [II]
- EPIPE　　パイプが壊れている [II]
- ERANGE　　数学関数の演算結果が，表現できる値の範囲を越えた
- EROFS　　読み込み専用のファイルシステムである
- ESPIPE　　キャラクタデバイスに対してシークしようとした
- ESRCH　　指定したプロセスが見つからない
- ETXTBSY　　実行ファイルが使用中である [II]
- EXDEV　　ファイルシステムをまたがってファイルを移動しようとした
- EWOULDBLOCK　　デッドロックが起きてしまう [II]

# D ................................ *LIBC*が見る環境変数

*LIBC*の提供する関数群のなかには，**環境変数**を参照して動作を変化させるものがいくつかあります。現在のバージョンの *LIBC*は，次の環境変数を参照しています。

- path

  spawnlp 関数や execlp 関数など，外部プログラムを実行させる関数は環境変数 path を参照し，指定されたプログラムを path ディレクトリから検索します。詳細は execlp 関数，あるいは spawnlp 関数を参照してください。当然ですが，この環境変数は必ず設定しておく必要があります。

- temp

  tmpnam 関数など，テンポラリファイルを作成する関数は環境変数 temp を参照し，指定されたディレクトリにテンポラリファイルを作成します。詳細は tempnam 関数を参照してください。この環境変数は必ず設定しておく必要があります。

- USER
  LOGNAME

  *LIBC*では getlogin 関数など，ユーザ名/ユーザログイン名を取得する関数を提供していますが，**Human68k** にはこのような概念がありません。そこで *LIBC*はこれらのユーザ名を取得する場合，環境変数 USER，LOGNAME を参照します。値は環境変数 USER の設定が優先されますが，もし USER が未定義ならば，環境変数 LOGNAME の値を使用します。また，LOGNAME も未定義ならば固定的に“root”が使用されます。
  これらの環境変数は自分で直接設定してもよいですが，たとえば“bash.x”や **ITA ToolBox** の“fish.x”を“login.x”とともに利用する場合は，自動的に設定されます。詳細は getlogin 関数を参照してください。
  この環境変数は，必要がなければ設定する必要はありません。

- UID
  EUID

*LIBC*ではgetuid関数やgeteuid関数など，ユーザID/実効ユーザIDを取得する関数を提供していますが，**Human68k**にはこのような概念がありません。そこで*LIBC*はこれらの値として環境変数UID/EUIDを参照します。それぞれUIDがユーザID，EUIDが実行ユーザIDを表します。もし環境変数EUIDが未定義ならば，実行ユーザIDはユーザIDと等しくなります。また環境変数UIDも未定義ならば，ユーザIDは0 (root)となります。

これらの環境変数は自分で直接設定することもできますが，“fish.x”，“bash.x”などのシェルが自動的に設定します。自分で設定する場合には，ユーザ名の設定との整合性が損なわれないように注意してください。詳細はgetuid関数，geteuid関数を参照してください。この環境変数は，必要がなければ設定する必要はありません。

- GID
  EGID

*LIBC*ではgetgid関数やgetegid関数など，グループID/実効グループIDを取得する関数を提供していますが，**Human68k**にはこのような概念がありません。そこで*LIBC*はこれらの値として環境変数GID/EGIDを参照します。それぞれGIDがグループID，EGIDが実行グループIDを表します。もし環境変数EGIDが未定義ならば，実行グループIDはグループIDと等しくなります。また環境変数GIDも未定義ならば，グループIDは0 (root)となります。

これらの環境変数は自分で直接設定することもできますが，“fish.x”，“bash.x”などのシェルが自動的に設定します。自分で設定する場合には，ユーザ名の設定との整合性が損なわれないように注意してください。詳細はgetgid関数，getegid関数を参照してください。この環境変数は，必要がなければ設定する必要はありません。

- SYSROOT

パスワードファイル(“/etc/passwd”)とグループファイル(“/etc/group”)は名前が固定のファイルです。通常，これらのファイルは“A:/”から読み込まれますが，環境変数SYSROOTを設定することで読み込む位置を変更することができます。たとえば環境変数SYSROOTに“B:/user”を設定すると，パスワードファイルは“B:/user/etc/passwd”が読み込まれます。詳細は，

getpwnam 関数あるいは getgrnam 関数などを参照してください。この環境変数は，必要がなければ設定する必要はありません。

- SHELL
  SYSTEM_SHELL

system 関数で，外部プログラムを起動する場合に使用するシェル (コマンドインタプリタ，たとえば “COMMAND.X” など) を指定します。もしこれが設定されていない場合，*LIBC*は “COMMAND.X” をシェルとして使用します。

環境変数 SHELL は自分で直接設定することもできますが，“fish.x”，“bash.x” などのシェルが自動的に設定します。system 関数が使用するシェルだけを変更したければ，環境変数 SYSTEM_SHELL のほうだけを変更してください。詳細は system 関数を参照してください。この環境変数は，必要がなければ設定する必要はありません。

- SHELL_OPT
  SYSTEM_SHELL_OPT

system 関数で外部プログラムを起動する場合に，シェルに対して引数を渡すために用いられるオプションを指定します。もしこれが設定されていない場合，*LIBC* は使用するシェルの形式によって自動的にオプションを選択します。この環境変数は，必要がなければ設定する必要はありません。

- SHELL_TYPE
  SYSTEM_SHELL_TYPE

system 関数で外部プログラムを起動する場合に，使用するシェル (コマンドインタプリタ) のタイプを指定します。タイプには，**Human68k** の “COMMAND.X” 形式のものと **UNIX** ライクな，いわゆる「シェル」形式のものと 2 種類あります。詳細は system 関数を参照してください。この環境変数は，必要がなければ設定する必要はありません。

- limit_core

*LIBC*でシグナルライブラリを用いる場合，バスエラーやアドレスエラーでコアダンプ状態になることがあります[1)]。このとき，環境変数 limit_core が設定されていないか，その値が 0 ならば，コアファイルは作成されません。また任意の値を指定することで，コアファイルのサイズを制限することができます。なお無制限にしたい場合は，−1 を設定してください。

1) 詳細については Vol.2 signal 関数 (P.289) を参照してください。

# E……………………………… *LIBC*とフリーウェア

*LIBC*は公開されているさまざまなフリーウェアを積極的に利用し，より豊富な機能を提供しています。現在のバージョンの*LIBC*では次のようなフリーウェアに対応しており，プログラム中からその機能を利用することができます。

- `TwentyOne`　　`TwentyOne` は**Human68k**にパッチを当て，ファイル名を21文字認識させたり，複数のピリオドを利用できるようにする常駐ソフトウェアです。*LIBC*はもちろんファイル名はすべて認識しますし，複数ピリオドについても正しく扱えるようになっています。ただし，`TwentyOne` の「大文字小文字を区別する」機能は用いないでください。*LIBC*はファイル名の比較が必要な場合，大文字と小文字を同じものとして認識していますから，この機能を利用すると動作がおかしくなります。また `TwentyOne` が常駐しているかどうかは，*LIBC*の動作と関係ありません。

- `lndrv`　　**lndrv** は，**Human68k** に **UNIX** と同じシンボリックリンクの機能を付加する常駐ソフトウェアです。*LIBC*はこの機能を利用することで，`lstat` 関数などのシンボリックリンクを正しく操作することができます。現在のバージョンの*LIBC*は **lndrv** の ver.2.12 を基に開発したものですから，将来 **lndrv** が大きく変更された場合，その動作に対する保証はありません。なお **lndrv** が常駐していない場合，`lstat` 関数は `stat` 関数と等しくなり，またシンボリックリンクを操作する関数はエラーを返します。

- `execd`　　`execd` はファイルに「実行属性ビット」を加えることで，“`.X`” や “`.R`” などの拡張子のないファイルもコマンドラインから実行できるようにする常駐ソフトウェアです。*LIBC*はこの実行属性ビットを正しく認識しますので，たとえば `stat` 関数などで実行可能ファイルかどうかを識別することができます。`execd` が常駐しているかどうかは，*LIBC*の動作に関係ありません。*LIBC*はファイルにつけられたビット情報のみを参照しています。

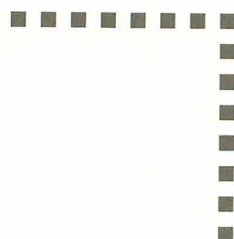
● HUPAIR 規格

HUPAIR 規格とは，実行させようとするプログラムに対して，長い引数列を渡すことができるようにと考えられた引数の引き渡し手順の規格です。従来のインタフェイスでは，実行するプログラムに対して，255 バイトまでしか引数を渡すことができませんでした。*LIBC*は，渡された引数を受け取る部分，また子プロセスとして外部プログラムを実行する際にそのプログラムに引数を渡す部分，これら両方とも HUPAIR 規格に準拠しています。したがって HUPAIR 規格に準拠した他のプログラムとの間で，長い引数列をやりとりすることができます。しかし HUPAIR 規格に準拠していないプログラムとの間は，従来どおりのインタフェイスで引数を渡します。

# F ……………………………………… *LIBC*用語一覧

最後に，*LIBC*で使用している用語について簡単に説明します。これ以外にもハードウェアに関連する専門用語が多数でてきますが，それらについてはここでは取り上げません。専門書を参考にしてください。

| 用語 | 説明 |
|---|---|
| ● DOS ファイルアトリビュート | ディスクに記録されているファイルの属性ビット (8ビット) |
| ● G フォーマット | printf 関数で用いるフォーマットで，F フォーマットと E フォーマットの中間の形式 |
| ● HUPAIR エンコード | HUPAIR 規格に準拠した引数列のまとめ方の手順 |
| ● HUPAIR 識別子 | HUPAIR 規格に準拠しているかどうかを表すフラグで，プログラムの実行開始位置+4バイトから8バイト "#HUPAIR\0" |
| ● Julian 日付 | 1月1日を0とし，年始めからの通算日で数える日付 |
| ● アラインメント | データの並び方に関する制限。たとえば，MC68000 は奇数バイトからのワードデータの読み出しはできない |
| ● アラームシグナル | 指定した時間が経過した時点で発生させることができるアラーム予約方式のシグナル |
| ● エポックタイム | 協定世界時 1970年1月1日0時0分0秒 |
| ● エラー指示子 | stdio ライブラリのファイルストリームに設定され，ストリームがエラー状態であることを示す |
| ● オープンモード | ファイルをオープンするときに指定するモードで，大別すると「読み込みのみ」,「書き込みのみ」,「両方」の3種類がある |
| ● 親プロセス | 子プロセスを実行したプログラム |
| ● 改行文字 | LF (0x0a) だが，処理系によっては CR (0x0d) の場合もある |

- 拡張 **UNIX** ファイルモード　**Human68k** の DOS ファイルアトリビュートと **UNIX** のファイルモードを両方扱えるように拡張した ***LIBC***独自のファイルモード表現
- 仮想ユーザ ID　ユーザ ID という概念がない **Human68k** 上で仮想的にユーザ ID を表現したもの
- 空文字列　長さが 0 の文字列 “” のこと
- カレントシグナルマスク　現在のシグナルマスクの設定値
- カレントプロセス　現在実行中のプログラム
- キャラクタデバイス　コンソール “CON”，シリアル (RS-232C) “AUX” など文字単位の操作しか行えないデバイス
- クイックソート　データのソートを行うアルゴリズムの一種
- グループ ID　**UNIX** 用語で，ユーザの所属するグループにつけられた ID 番号
- グループファイル　**UNIX** 用語で，グループの一覧を納めたファイル
- コアダンプ　**UNIX** 用語で，実行中のプログラムのある一瞬のメモリ状態をファイルに保存すること
- コアファイル　**UNIX** 用語で，コアダンプによって作成されるファイル
- 子プロセス　現在実行中のプログラムから呼び出される外部プログラム
- コンソール　画面
- シェル　**UNIX** 用語で，コマンドインタプリタのこと
- シグナルセット　シグナルの一覧表のようなもの
- シグナルの配信　シグナル割り込みを発生させること
- シグナルハンドラ　シグナル割り込みを処理する関数
- シグナルペンディングセット　割り込みが発生しても，アプリケーションに配信されていないシグナルの一覧表
- シグナルマスク　割り込みが発生してもアプリケーションに配信されないようにするシグナルの一覧表
- 指数形式　“1.57863e-67” のような正規化された数値表現
- システムエラーコード　**Human68k** の DOS コールが返してくるエラーコード

- **システム制限値**　メモリの最大容量など，アプリケーションに許されるさまざまな最大値
- **実行属性ビット**　フリーウェアであるexecdが使用するDOSファイルアトリビュートのなかの1ビット
- **実時間**　現実世界の時間
- **ジャンプポイント**　setjmp関数で記憶されたプログラム中の任意の位置
- **終端指示子**　stdioライブラリのファイルストリームに設定され，ストリームがファイルの終端に達したことを示す
- **終了コード**　プログラムの実行結果として親プロセスに渡される整数値
- **詳細時間**　年月日，時分秒の形式で表現した時間
- **シンボリックリンク**　別のファイルへの道順が書いてあるファイルで，このファイルをアクセスすると，その道順にそって別のファイルにアクセスされる
- **スタックフレーム**　関数に渡された引数や，その関数で使用するローカル変数などを記憶しておくスタック領域のなかの一領域
- **スタック領域**　関数との間の引数の受け渡しやローカル変数のために使用されるメモリ領域
- **大域ジャンプ**　longjmp関数とsetjmp関数を用いて行う，関数の枠を越えた処理の分岐
- **タイムゾーン**　時間帯
- **タイムゾーン情報**　時間帯の情報で，「時差」，「夏時間の有無」，「時間帯の名称」など
- **端末デバイス**　キャラクタデバイス
- **地域時間**　地域のローカルな時間
- **地域時間情報**　タイムゾーン情報
- **ディレクトリエントリ**　ファイルのサイズ/変更時間，ディスク内の位置などを記憶しているデータ
- **ディレクトリストリーム**　ディレクトリファイルのなかのディレクトリエントリのデータ列を，ストリームとして表現したもの
- **テキストモード**　1行の終わりを示すLFをCR，LFの2バイトとして扱うモード

- デリミタ　区切り記号
- トークン　区切り記号で区切られた部分文字列
- バイナリサーチ　データの検索を行うアルゴリズムの一種
- バイナリモード　1行の終わりを示すLFをそのまま1バイトで扱うモード
- パイプ　UNIX用語で，FIFO(先入れ先出し)方式でデータを入出力することができるデバイス
- パス区切り記号　パス名のなかのディレクトリ区切りを表現する記号で，UNIXでは“/”，Human68kやMS–DOSでは“\”
- パスワードファイル　UNIX用語で，ユーザ一覧を納めたファイル
- ビルトイン関数　GCCが出力するコードのなかに直接埋め込まれる実体のない関数。インライン関数の一種
- ファイルアクセスモード　ファイルの種類やアクセスの許可/不許可などを表したデータ
- ファイルストリーム　ファイルの中身をデータの連続体(ストリーム)として表現したもの
- ファイルハンドル　低水準ファイル入出力で，特定のファイルを表す識別子
- ファイルポインタ　ファイルの先頭から終わりまでで，現在見ている位置
- 物理ドライブ番号　Human68kが起動時に認識したドライブ番号
- ブレーク値　ヒープ領域の終端の位置
- プロセスID　UNIX用語で，現在実行中の特定のプロセスを表す識別子
- ページ　UNIX用語で，MMUを使った仮想記憶でOSが一度に扱うメモリ単位
- メモリ管理ポインタ　Human68kの管理するメモリブロックの先頭に記録されている管理情報
- メモリブロック　任意サイズのメモリ領域
- ユーザID　UNIX用語で，特定のユーザを表す識別子
- 読み込み禁止ファイル　存在するが，中身を見ることが禁止されているファイル

| | |
|---|---|
| ● 乱数シード | 乱数を発生させる乱数形列の基礎となるもの |
| ● リアルタイムクロック | **X68000** および **X68030** では RTC のこと |
| ● リエントラント | 再入可能 (実行環境と一切の関わりを持たない関数) |
| ● リンクカウント | **UNIX** 用語で，ハードリンクしたときに何か所からリンクされているかを記録するカウンタ |
| ● 暦時間 | エポックタイムからの通算秒数で時間を表現する方法 |
| ● レジスタ渡し | 関数間の引数のやりとりを，スタックではなくレジスタを介して行う方法 |
| ● 論理ドライブ番号 | 物理ドライブを，仮想ドライブなどの機能を用いて変更した後のドライブ番号 |

Chapter 4

# Appendix B

---

Appendix B として，*LIBC* を作成するうえで参照した Human68k の内部情報を掲載します。ただし，この資料は公式な資料ではありませんから，記載された情報が正しいとはかぎりません。また，Human68k のバージョンによっても異なりますので，あくまでも参考程度にとどめておいてください。

# A……………………… システムのエラーコード

最初に**Human68k**のDOSコールが返すエラーコードの一覧表とSCSI/IOCSコールが返すエラーコードの一覧表を掲載します。

## ○ DOSコールのエラーコード

| コード | 解説 |
|---|---|
| -1 | 無効なファンクションコールを実行した |
| -2 | 指定したファンクション名は見つからない |
| -3 | 指定したディレクトリが見つからない |
| -4 | これ以上ファイルをオープンできない |
| -5 | ディレクトリやボリュームラベルは指定できない |
| -6 | 不正なファイルハンドルを指定した |
| -7 | メモリ管理領域が破壊された |
| -8 | メモリが足りなくなった |
| -9 | 無効なメモリ管理ポインタを指定した |
| -10 | 不正な環境を指定した |
| -11 | 実行ファイルのフォーマットがおかしい |
| -12 | 不正なファイルアクセスモードを指定した |
| -13 | 不正なファイル名を指定した |
| -14 | 不正な引数を指定した |
| -15 | 不正なドライブ番号を指定した |
| -16 | カレントディレクトリは削除できない |
| -17 | IOCTRLできないデバイスを指定した |
| -18 | これ以上ファイルが見つからない |
| -19 | このファイルには書き込みできない |
| -20 | すでに存在するディレクトリを指定した |
| -21 | ディレクトリが空でないので削除できない |
| -22 | ディレクトリが空でないのでリネームできない |
| -23 | ディスクが一杯になった |
| -24 | これ以上ファイルを作成できない |
| -25 | 指定された位置にはシークできない |
| -26 | 多重にスーパーバイザモードに入ろうとした |

| コード | 解 説 |
|---|---|
| -27 | 未使用 |
| -28 | 同名のスレッドがすでに存在する |
| -29 | メッセージが受け取られなかった |
| -30 | 不正なスレッド番号を指定した |
| -31 | 未使用 |
| -32 | シェアリング可能なファイル数を越えた |
| -33 | ロック違反 |

## ○ SCSI のエラーコード

### ● 上位 INTS(SPC の割り込み原因)

| ビット | 意 味 |
|---|---|
| 16 | RESET コンディション割り込み |
| 17 | SPC ハードウェア割り込み |
| 18 | セレクションタイムアウト割り込み |
| 19 | 実行しようとする転送フェーズと SCSI 上での要求されている転送フェーズとが一致しなかったか，あるいは転送実行中に他の転送フェーズが要求されてきたときの割り込み |
| 20 | SPC に対するコマンド終了割り込み |
| 21 | DISCONNECTED 割り込み |
| 22 | RESELECTED 割り込み |
| 23 | SELECTED 割り込み |

各ビットは，0 で割り込みなし，1 で割り込みありを表す。

### ● 下位 PSNS(SCSI バス上の制御信号の状態)

| ビット | 意 味 |
|---|---|
| 0 | I/O (データの方向を示す信号) |
| 1 | C/D (コマンドかデータフェーズかを示す信号) |
| 2 | MSG (メッセージフェーズを示す信号) |
| 3 | BSY (SCSI バスの使用中を示す信号) |
| 4 | SEL (選択信号) |
| 5 | ATN (アテンション条件を示す信号) |
| 6 | ACK (データ転送許可信号) |
| 7 | REQ (データ転送要求信号) |
| 8 | 0 |
| 9 | 0 |

各ビットは，信号が 0 でノンアクティブ，1 でアクティブであることを表す。

# B……………………… Human68k の内部情報

*LIBC*を作成するにあたって，我々が収集した**Human68k** の内部情報をわかっているかぎり掲載します。ただし，この情報はいろいろな人が解析した結果をまとめたものであり，公式な資料として公開されているものではありません。したがって，ここに記載されていることはあくまでも参考程度に考えてください。また，この情報はほとんど **Human68k** ver.2 を基に作成されたものなので，**Human68k** ver.3 では異なることもあります。

また，**Human68k** ver.2.03(92 年版) とは，起動メッセージの "Copyright" 部分に "92" と表示されるバージョンのことを言います。

## ○ Human68k のワークエリアの内部情報

| アドレス | 解 説 |
|---|---|
| $1800-$1bff | DOS コールベクタテーブル (各 4 バイト)<br>$1bc0(L) DOS コール $fff0 retshell<br>$1bc4(L) DOS コール $fff1 ctlabort<br>$1bc8(L) DOS コール $fff2 errabort<br>$1be0(L) DOS コール $fff8 open_pr<br>$1be4(L) DOS コール $fff9 kill_pr<br>$1bfc(L) DOS コール $ffff change_pr<br>上記ベクタに関しては処理をのっとるのではなく，ユーザ処理ルーチンが本来の処理に加えてコールされるだけである |
| $1c00(L) | 現在のプロセスにおけるメモリ最終アドレス+1 |
| $1c04(L) | 現在のプロセスにおけるメモリ先頭アドレス |
| $1c08(W) | inDOS フラグ |
| $1c0a(B) | 実行中の DOS コールファンクションナンバー (非マルチタスク環境下では，inDOS フラグが 0 から 1 に変わるタイミングでしかセットされない) |
| $1c0b(B) | NEWFAT= で指定した値 (初期値：$6815 の内容) |
| $1c0c(W) | IOCTRL(11) 第 1 パラメータ (初期値：3) |
| $1c0e(W) | IOCTRL(11) 第 2 パラメータ (初期値：100) |
| $1c10(W) | VERIFY フラグ |

| アドレス | 解 説 |
|---|---|
| $1c12(B) | BREAK フラグ |
| $1c13(B) | CTRL+P フラグ |
| $1c14(B) | プロセス切り替えのタイミングであることを示すフラグ |
| $1c15(B) | カレントドライブ番号 |
| $1c16(B) | trap #11 STOP キー<br>bit7 と bit0 が変化するが，詳細は不明 |
| $1c17(B) | trap #10 リセット/パワーオフの hook に入ったときに 1 になるフラグ。これが 1 だと終了処理後リセット? |
| $1c18(L) | trap #10 リセット/パワーオフの hook に入ったときの d0 退避領域 (リセット/パワー OFF 判定フラグ?) |
| $1c1c(L) | デバイスチェインの最後のデバイスヘッダへのポインタ |
| $1c20(L) | HUMAN.SYS の MCB アドレス (メモリチェインの先頭) |
| $1c24(L) | HUMAN.SYS のメモリブロックがどこまで使われているかを示す。スーパーバイザ領域はこの値を基に 8K バイトバンダリで設定される |
| $1c28(L) | 現在のプロセスの PSP アドレスを格納してあるワークを指すポインタ (**Human68k** ver.2.03–92 年版だと $12b54 で，$12b54 には現在のプロセスの PSP アドレスが格納されている) |
| $1c2c(L) | 標準 FCB 以外 (ファイルハンドルが 5 以上) の FCB インデックステーブルへのポインタ (バッファの終端+1) |
| $1c30(L) | FCB テーブルへのポインタ (予備 FCB インデックステーブルの終端+1) |
| $1c34(L) | |
| $1c38(L) | カレントディレクトリテーブルへのポインタ |
| $1c3c(L) | 物理デバイス情報テーブルへのポインタ |
| $1c40(L) | SHARE= 管理構造体の先頭アドレス |
| $1c44(L) | COMMON= ポインタ |
| $1c48(L) | COMMON= ポインタ |
| $1c4c(L) | COMMON= ポインタ (バッファの終端?) |
| $1c50(L) | プロセス構造体テーブルトップへのポインタ |
| $1c54(L) | 現在のプロセスのプロセス構造体へのポインタ |
| $1c58(W) | 最大プロセス数 ( PROCESS= の第 1 引数で指定した値−1) |
| $1c5a(W) | 現在生成されているプロセスの数 |
| $1c5c(L) | DOS コール時のベクタブランチ直前の a7 の値 ( inDOS フラグが 0 から 1 に変わるタイミングでしかセットされない) |
| $1c60(W) | アボート時の SR |
| $1c62(L) | アボート時の SSP |
| $1c66(L) | デフォルトの trap #11 処理ルーチンへのポインタ |
| $1c6a(L) | デフォルトの trap #10 処理ルーチンへのポインタ (ROM を指しており，jmp させると IPL する) |
| $1c6e(W) | 最大ハンドル番号 ( FILES= 設定値+2) |

| アドレス | 解 説 |
|---|---|
| $1c70(W) | BUFFERS= 第2パラメータ(初期値：$6804の内容) |
| $1c72(B) | BUFFERS= 第1パラメータ |
| $1c73(B) | 最終ドライブのドライブ番号(LASTDRIVE= の内容) |
| $1c74(B) | ドライブ数関連(初期値：$6807の内容) |
| $1c75(B) | 接続されているドライブ数? |
| $1c76(W) | SHARE= ファイル数(初期値：$680aの内容で93まで) |
| $1c78(W) | SHARE= 領域数(初期値：$680cの内容で266まで) |
| $1c7a(L) | SHARE= 管理バッファのサイズ |
| $1c7e(26B) | ドライブ配置テーブル(ドライブ番号変換用) |
| $1c98(L) | OPENしたFCBのバッファフラッシュ関係のポインタ? |
| $1c9c(L) | OPENしたファイルハンドルのバッファフラッシュ関係? |
| $1ca0(B) | EXEC関連フラグ |
| $1ca1(B) | EXECのモード |
| $1ca2(B) | INSキーのON/OFFフラグ |
| $1ca3(B) | trap #14する前に0，後に−1? |
| $1ca4(L) | GETCバッファ読み込みポインタ? |
| $1ca8(W) | GETCバッファ残りカウンタ? |
| $1caa(L) | EXEC関連ポインタ(MCB+$100) |
| $1cae(L) | プロセスの終了コード |
| $1cb2(L) | 最後に実行したコマンドライン文字列へのポインタ |
| $1cb6(L) | CLOCKデバイスのデバイスヘッダへのポインタ |
| $1cba-$67ff | **Human68k** ver.2.03までは未使用領域のはず |
| $6800 | HUMAN.SYS開始アドレス(OSスタック下限?) |

### ○ Human68kの内部デフォルト値

| アドレス | 解 説 |
|---|---|
| $6802(B) | FILES (15) |
| $6803(B) | BUFFERSバッファ数 (20) (2 ～ 249) |
| $6804(W) | BUFFERSバッファサイズ (1024) (1024 ～ 32768) |
| $6806(B) | LASTDRIVE番号 (25) |
| $6807(B) | DRIVES関連 (25) |
| $6808(B) | BREAKフラグ (0) |
| $6809(B) | VERIFYフラグ (0) |
| $680a(W) | SHAREファイル数 (0) |
| $680c(W) | SHARE領域数 (0) |
| $680e(L) | COMMON領域サイズ (0) −1024 (1024Kバイト) |
| $6812(B) | PROCESS第1引数 (0) |
| $6813(B) | PROCESS第2引数 (0) |

| アドレス | 解 説 |
|---|---|
| $6814(B) | PROCESS 第 3 引数 (0) |
| $6815(B) | NEWFAT (0) |

以降，表中で “$A:$B” と表記しているのは，“$A” が **Human68k** ver.2.02 でのアドレス，“$B” が **Human68k** ver.2.03(92 年版) でのアドレスを意味している。ただし **Human68k** ver.2.03(91 年版) の場合は，92 年版から−2 したアドレスになる。

| アドレス | 解 説 |
|---|---|
| $?????:$07d50 | HUMAN.SYS の MCB |
| $?????:$07d58 | HUMAN.SYS のメモリブロックの終端アドレス (スーパーバイザ領域のリミットアドレス) |
| $?????:$07d60 | HUMAN.SYS の PDB |

## ○ マルチタスク関連の内部情報

| アドレス | 解 説 |
|---|---|
| $?????:$0e914(W) | プロセス切り替えが，DOS コールによるものであるのか，割り込みによるものかのフラグ (0 で DOS コール) |
| $?????:$0e916(L) | NMI ベクタ保存ワーク |
| $?????:$0e91a(L) | タイムカウンタ (10ms ごとに+1) |
| $?????:$0e91e(L) | |
| $?????:$0e922(B) | レベルカウンタ |
| $?????:$0e923(B) | レベルカウンタのインターバル値 |
| $?????:$0e924(L) | プロセス間通信バッファへのポインタ |
| $?????:$0e938(16B) | プロセス名 |
| $?????:$0df4c | Timer–D 割り込みハンドラのエントリアドレス |
| $?????:$0df88 | DOS コール終了のタイミングでの change_pr |
| $?????:$0e122 | NMI 割り込みハンドラのエントリアドレス (kill_pr と change_pr の実行中) |
| $?????:$0e12a | NMI 割り込みハンドラのエントリアドレス |
| $?????:$0e680 | IOCS TIMERDST hook のエントリアドレス |
| $?????:$0e766 | trap #14 エラー表示ハンドラのエントリアドレス |
| $?????:$0e6a0(B) | inNMI フラグ |
| $?????:$0e6a1 | .even |
| $?????:$0e6a2(L) | NMI ベクタの保存ワーク |
| $?????:$0e6a6(L) | time_pr の戻り値 |
| $?????:$0e6aa(L) | 元の時間 |

| アドレス | 解 説 |
|---|---|
| $?????:$0e6ae(B) | タイムスライスレベル(現在値) |
| $?????:$0e6af(B) | タイムスライスレベル(初期値) |
| $?????:$0e6b0 | HUMAN.SYS のプロセス間通信バッファ構造体 |
| $?????:$0e6c4 | HUMAN.SYS のプロセス名 |

### ○ 内蔵デバイスドライバの情報

| アドレス | 解 説 |
|---|---|
| $?????:$0eac2 | NUL デバイスエントリ |
| $?????:$0eb5a | CON デバイスエントリ |
| $?????:$0f6e6 | AUX デバイスエントリ |
| $?????:$0f7c0 | PRN デバイスエントリ |
| $?????:$0f82a | LPT デバイスエントリ |
| $?????:$0f8e0 | CLOCK デバイスエントリ |
| $?????:$0f9d0 | DISK2HD デバイスエントリ |

### ○ データ領域の内部情報

| アドレス | 解 説 |
|---|---|
| $?????:$0fedc(L) | malloc ポインタ |
| $?????:$0fee0(B) | |
| $?????:$0fee1(B) | |
| $?????:$0feee(256B) | デフォルトのシェル "COMMAND /P" |
| $?????:$0ffdf(S) | |
| $?????:$0fffa(S) | |
| $?????:$10009(S) | |
| $?????:$10027(S) | |
| $?????:$10029(S) | |
| $?????:$10038(S) | "^C" |
| $?????:$1003a(S) | "\r\n\0" |
| $?????:$1003e(4B) | |
| $?????:$10042(S) | "path=" |
| $?????:$10048(B) | |
| $?????:$10049(S) | EXEC で検索する拡張子テーブル |
| $?????:$10050(S) | EXEC で検索する際に補完する ".*" |
| $?????:$10054(B) | |
| $?????:$10055(9B) | |

### ○ エラー表示ハンドラで使用される領域

| アドレス | 解説 |
|---|---|
| $?????:$1005e(2W) | エラーメッセージ座標 (X,Y) |
| $?????:$10062(2W) | エラーメッセージ座標 (X,Y) |
| $?????:$10066(2W) | エラーメッセージ座標 (X,Y) |
| $?????:$1006a(2W) | エラーウィンドウ座標 (X,Y) |
| $?????:$1006e(S) | エラーメッセージ |
| $?????:$10078(S) | .. |
| $?????:$10087(S) | .. |
| $?????:$1008d(S) | .. |
| $?????:$10093(S) | .. |
| $?????:$10096(S) | .. |
| $?????:$100cb(S) | Bus error |
| $?????:$100e8(S) | Address error |
| $?????:$10105(S) | Undefined instruction |
| $?????:$10122(S) | Devision by zero |
| $?????:$1013f(S) | CHK execution |
| $?????:$1015c(S) | TRAPV execution |
| $?????:$10179(S) | Supervisor command |
| $?????:$10196(S) | Interrupt |
| $?????:$101b3(S) | Float package not installed |
| $?????:$101d0(S) | Illeagl unit number |
| $?????:$10205(S) | Disk not ready |
| $?????:$1023a(S) | Illegal device command |
| $?????:$1026f(S) | CRC error |
| $?????:$102a4(S) | Broken FAT |
| $?????:$102d9(S) | Seek error |
| $?????:$1030e(S) | Invalid media |
| $?????:$10343(S) | Sector not found |
| $?????:$10378(S) | Printer not ready |
| $?????:$103ad(S) | Write error |
| $?????:$103e2(S) | Read error |
| $?????:$10417(S) | Error occured |
| $?????:$1044c(S) | Disk protected |
| $?????:$10481(S) | Not writetable |
| $?????:$104b6(S) | Sharing error |
| $?????:$104ec(W) | 保存/復元されるコントラスト値のワーク |
| $?????:$104ee(16W) | 保存/復元されるテキストパレット値のワーク |
| $?????:$1050e(W) | アボートを選択すると1になる。パレットの復元をするかどうかのフラグとして利用 |

| アドレス | 解 説 |
|---|---|
| $?????:$10510(4W+L) | keep(x,y), set(x,y), size |
| $?????:$1051c(1664B) | テキストパレット0入出力バッファ |
| $?????:$10b9c(1664B) | テキストパレット1入出力バッファ |
| $?????:$1121c(10B) | 10進/16進変換用ワーク |

## ○ CONデバイスで使用される領域の情報

| アドレス | 解 説 |
|---|---|
| $?????:$11226(B×15) | チェックテーブル (knjctrl) |
| $?????:$11235(5B) | チェックテーブル (knjctrl) |
| $?????:$1123a(5B) | チェックテーブル (knjctrl) |
| $?????:$11250(L) | |
| $?????:$11254(32B) | |
| $?????:$11274(W) | |
| $?????:$11276(W) | sftsnsのステータス |
| $?????:$11278(8B) | rmacnvのa1 |
| $?????:$11280(8B) | rmacnvのa2 |
| $?????:$11288(B) | |
| $?????:$11289(B) | |
| $?????:$1128a(1536B) | テキストパレット0の入出力バッファ (hendsp) |
| $?????:$1188a(1536B) | テキストパレット1の入出力バッファ (hendsp) |
| $?????:$11e8a(1280B) | テキストパレット0の入出力バッファ (hendsp) |
| $?????:$1238a(1280B) | テキストパレット1の入出力バッファ (hendsp) |
| $?????:$1288a(B) | conctrl |
| $?????:$1288b(B) | conctrl |
| $?????:$1288c(712B) | function key define data |
| $12dc8:$12b54(L) | カレントPSPポインタ |
| $12dcc:$12b58(12B) | ファイルハンドルの使用/未使用のビットマップ |
| $12dd8:$12b64(5W) | FCB関連 |
| $12de2:$12b6e(5W) | 標準ファイルハンドル(0～4)のFCBインデックステーブル |
| $12dec:$12b78(W) | ファイルハンドル5のFCBインデックス |
| $12dee:$12b7a(480B) | 標準ファイルハンドル(0～4)のFCB<br>$12b7a … stdin<br>$12bda … stdout<br>$12c3a … stderr<br>$12c9a … stdaux<br>$12cfa … stdprn |
| $12fce:$12d5a(96B) | ファイルハンドル5のFCB |

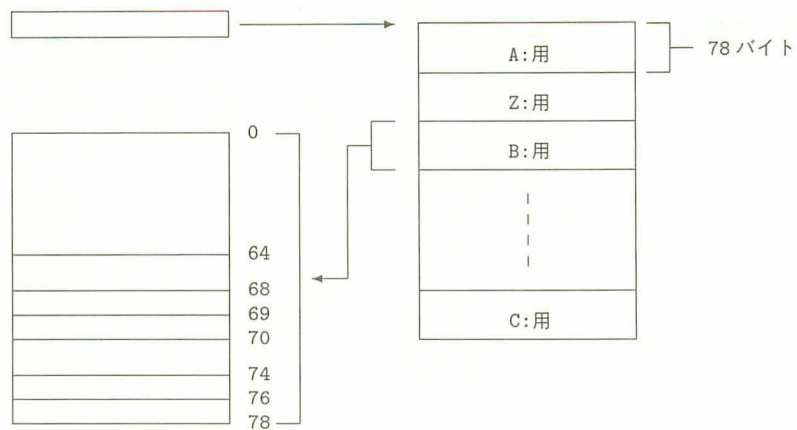
| アドレス | 解 説 |
|---|---|
| $1302e:$12dba(260B) | |
| $13132:$12ebe(B) | ブレークフラグ (ON/OFF/KILL) (ブート時の一時ワーク) |
| $?????:$12ec0(L) | bind ファイルオープンフラグ (クローズされると−1) |
| $?????:$12ec4(L) | DOS コール初期エントリテーブル |

### ○ FCB の内部情報

| オフセット | サイズ | 内 容 |
|---|---|---|
| $00 | B | リンクカウンタ (DUP されると 1 加算される) |
| +$01 | B | 装置情報 |
| | | キャラクタデバイスの場合<br>bit 0 ･･･ 1：標準入力デバイス<br>bit 1 ･･･ 1：標準出力デバイス<br>bit 2 ･･･ 1：NUL デバイス<br>bit 3 ･･･ 1：CLOCK デバイス<br>bit 4<br>bit 5 ･･･ 0：COOKED　1：RAW<br>bit 6<br>bit 7 ･･･ 1 |
| | | ブロックデバイスの場合<br>bit 0-4 ･･･：物理ドライブ番号 |
| +$02 | L | デバイスドライバへのポインタ (CHAR) |
| | | 内部 DPB へのポインタ (BLOCK) |
| +$06 | L | ファイルポインタの値 |
| +$0a | L | 排他制御情報へのポインタ |
| +$0e | B | アクセスモード |
| +$0f | B | ブロックデバイスの場合<br>ディレクトリエントリのセクタ内位置 (0 ～ 31) |
| | | キャラクタデバイスの場合<br>未使用 ($00) |
| +$10 | B | アクセス中のクラスタ中のセクタ |
| +$11 | B | |
| +$12 | W | アクセス中のクラスタ番号 |
| +$14 | L | アクセス中のセクタ番号 |
| +$18 | L | I/O バッファ先頭 |
| +$1c | L | ブロックデバイスの場合<br>ディレクトリエントリのセクタ番号 |
| | | キャラクタデバイスの場合<br>未使用 ($00000000) |
| +$20 | L | ファイルポインタ移動の上限 |

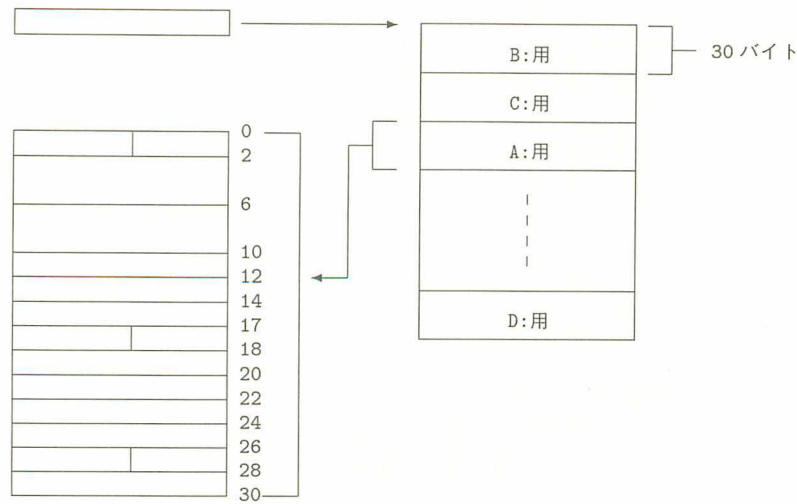
| オフセット | サイズ | 内 容 |
|---|---|---|
| +$20 | L | ファイルポインタ移動の上限 |
| +$24 | 8B | ファイル名1(あまった部分は$20) |
| +$2c | 3B | ファイル拡張子(あまった部分は$20) |
| +$2f | B | ファイル属性<br>bit 0 ··· 読み込み専用<br>bit 1 ··· 不可視<br>bit 2 ··· システム<br>bit 3 ··· ボリューム<br>bit 4 ··· ディレクトリ<br>bit 5 ··· 通常のファイル<br>bit 6<br>bit 7 |
| +$30 | 10B | ファイル名2(あまった部分は$00) |
| +$3a | W | 時刻<br>bit 15 ～ 11 ··· 時<br>bit 10 ～ 05 ··· 分<br>bit 04 ～ 00 ··· 秒×2 |
| +$3c | W | 日付<br>bit 15 ～ 09 ··· 年<br>bit 08 ～ 05 ··· 月<br>bit 04 ～ 00 ··· 日 |
| +$3e | W | 先頭FAT番号 |
| +$40 | L | ファイルサイズ |
| +$44 | 7L | 未使用($ffffffff) |

### ○ DOS コール時のベクタブランチ直前の a7 の値

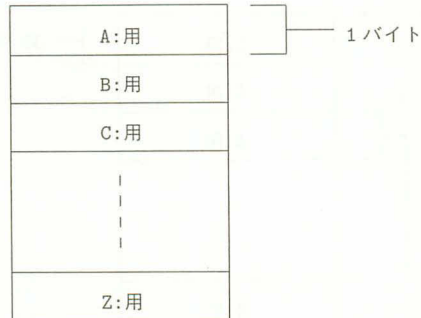
| アドレス | 解 説 |
|---|---|
| $1c5c(L) | DOS コール時のベクタブランチ直前の a7 の値 |

実際のスタックフレーム

| |
|---|
| d1 |
| d2 |
| d3 |
| d4 |
| d5 |
| d6 |
| d7 |
| a0 |
| a1 |
| a2 |
| a3 |
| a4 |
| a5 |
| a6 |
| a7 |

DOS コールの INTVCS などでベクトルをうばった処理ルーチンから，直接レジスタに値を返したい場合は，$1c5c からポインタを取り出し，該当する位置に値をセットしてリターンするだけでよい。あとは DOS が自動的に処理する。

### ○ カレントディレクトリテーブルの情報

| アドレス | 解 説 |
| --- | --- |
| $1c38(L) | カレントディレクトリテーブルへのポインタ |

| オフセット | 内 容 |
| --- | --- |
| +0 | 現在のカレントパスをフルパスで示す(仮想ドライブの場合はマウントされているドライブのディレクトリ名)。区切記号は$09が“\”の代わりに使われている。ターミネータはNULL |
| +64 | つねにNULL |
| +68 | つねに0 |
| +69 | 仮想ドライブ情報<br>$40 ··· リニアドライブ<br>$50 ··· 仮想ドライブ<br>$60 ··· 仮想ディレクトリ |
| +70 | 物理デバイス情報へのポインタ |
| +74 | つねに$ffff(ドライブZ:は$0000になっている) |
| +76 | つねに$0002 |

## ○ 物理デバイス情報テーブルの情報

| アドレス | 解 説 |
|---|---|
| $1c3c(L) | 物理デバイス情報テーブルへのポインタ |

| オフセット | 内 容 |
|---|---|
| +0.B | 装置番号 |
| +1.B | ユニット番号 |
| +2.L | デバイスヘッダへのポインタ |
| +6.L | 次のテーブルへのポインタ |
| +10.W | 1セクタ当たりのバイト数 |
| +12.W | 1クラスタ当たりのセクタ数−1 |
| +14.W | FAT の先頭セクタ番号 |
| +16.B | FAT 領域の数 |
| +17.B | 1個の FAT 領域に使用するセクタ数 |
| +18.W | ルートディレクトリのエントリ数 |
| +20.W | データ部先頭セクタ番号 |
| +22.W | 総クラスタ数+1 |
| +24.W | ルートディレクトリ先頭セクタ番号 |
| +26.B | メディアバイト |
| +28.B | |
| +30.W | つねに$0002 |

### ○ ドライブ配置テーブル(ドライブ番号変換用)

| アドレス | 解 説 |
|---|---|
| $1c7e(26B) | ドライブ配置テーブル(ドライブ番号変換用) |

ドライブ名に対して，現在実際に割り当てられているドライブ番号がそれぞれ格納されている(ドライブ A:=$00, B:=$02 ... Z:=$19)。“drive.x” で入れ替えると，このテーブルは変更される。故意に同じ番号をセットするとおもしろい動作をする。

### ○ _dos_s_process 関数により確保されるプロセスメモリ空間の構造

大もとのメモリブロックの属性が$fd になっていて，そのブロックを細切れに分けている。ヘッダや構造に関しては通常のメモリブロックと同様で，単にチェインがサブメモリで完結しているだけである。

### ○ メモリブロック属性

| コード | 意 味 |
|---|---|
| $ff | 常駐メモリブロック(KEEP) |
| $fe | MEMDRV? |
| $fd | サブメモリブロック? |

### ○ X形式実行ファイルの情報

| オフセット | 解 説 |
|---|---|
| +$00(W) | XMAGIC |
| +$02(W) | メモリ割り当てモード<br>%00 ・・・ 通常<br>%10 ・・・ 上位メモリ (malloc2(2))<br>%01 ・・・ 最小メモリ (malloc2(1)) |
| +$3c(L) | バインド情報までのオフセット |

ただし **Human68k** ver.3.00 まではバグがあり，メモリ割り当てモードのビット 1 (上位メモリへの割り付け) しか有効ではない。また，ビット 0 のビット位置に関しては推測でしかない。

### ○ Z形式実行ファイルの情報

| オフセット | 解 説 |
|---|---|
| +$00(W) | ZMAGIC ($601a) |
| +$02(L) | テキストサイズ |
| +$06(L) | データサイズ |
| +$0a(L) | BSS サイズ |
| +$16(L) | ロードアドレス |
| +$1a(W) | d5 |

合計$2c バイトである。

### ○ Human68k の common 領域

common 領域の最大サイズは 1024K バイトである。

# 参考文献

[ 1] *Application Environment Specification (AES) – Operating System Programming Interface Volume, Revision A.* Open Software Foundation, Prentice Hall, Inc., Englewood Cliffs, New Jersy.

[ 2] 『C言語による最新アルゴリズム辞典 (Software Technology 13)』 奥村晴彦著，技術評論社

[ 3] 『X68000用 C COMPILER PRO–68K ver1.01 プログラマーズマニュアル (SHARP COMPUTER SOFTWARE)』 シャープ

[ 4] 『X68000用 C COMPILER PRO–68K ver1.01 アセンブラマニュアル (SHARP COMPUTER SOFTWARE)』 シャープ

[ 5] 『X68000用 C COMPILER PRO–68K ver2.01 プログラマーズマニュアル (SHARP COMPUTER SOFTWARE)』 シャープ

[ 6] 『Inside X68000』 桑野雅彦著，ソフトバンク

[ 7] 『プログラマーのための X68000 環境ハンドブック』 吉沢正敏/市原昌文著，工学社

[ 8] P.J.Plauger. *THE STANDARD C LIBRARY.* Prentice Hall, Inc., Englewood Cliffs, New Jersy.

[ 9] 『ANSI C 言語辞典』 平林雅英著，技術評論社

[10] Donald Lewine. *POSIX, Programmer's Guide, Writing Portable UNIX Programs.* O'Reilly & Associates, Inc.

[11] *OSF/1(TM) Operating System Programmers' Reference.* Open Software Foundation, Prentice Hall, Inc., Englewood Cliffs, New Jersey.

[12] W. Richard Stevens. *Advanced Programming in the UNIX Environment, ADDISON WESLEY PROFESSIONAL COMPUTING SERIES.* Addison Wesley.

[13] 『ポケット版 MS–C Ver.6.0 関数リファレンス』 高瀬典明著，ソフトバンク

[14] 『UNIX ネットワークプログラミング』 W. リチャード. スティーブンス著，篠田陽一訳，Prentice Hall, Inc., Englewood Cliffs, New Jersy，トッパン

[15] 『プログラミング言語 C 第 2 版』 B.W. カーニハン/D.M. リッチー著，石田晴久訳，Prentice Hall, Inc., Englewood Cliffs, New Jersy，共立出版

[16] 『MINIX オペレーティングシステム — OPERATING SYSTEM DESIGN AND IMPLEMENTATION』 ANDREW S.TANENBAUM 著，坂本文監修，大西照代訳，Prentice Hall, Inc., Englewood Cliffs, New Jersy，アスキー

[17] 『C ユーザのための ANSI C 言語大辞典』 マーク・ウィリアムズ社編，坂本文/今泉貴史監訳，パーソナルメディア

[18] 『数値演算入門』 大貫広幸著，CQ出版

[19] 『C による科学技術計算』 小池慎一著，CQ出版

[20] 『M68000 マイクロプロセッサ ユーザーズ・マニュアル』 MOTOROLA, Prentice Hall, Inc., Englewood Cliffs, New Jersy，CQ出版

[21] *M68000 FAMILY REFERENCE (68000–68040)*, MOTOROLA

[22] 『HD68000，HD68HC000 MPU (Micro Processing Unit)』 日立製作所

[23] *MC68010/MC68012 16/32 BIT VIRTUAL MEMORY MICROPROCESSORS*, MOTOROLA

[24] 『MC68030 ユーザーズ・マニュアル (ENHANCED 32 BIT MICROPROCESSOR)』 日本モトローラ，CQ出版

[25] 『スーパーマイクロプロセッサ (bit 臨時増刊 1991–3 Vol.23 No.4)』 高澤嘉光/飯島純一著，共立出版

[26] 『SCSI ボード (CZ–6BS1) 取扱説明書』 シャープ

[27] 『最新 SCSI マニュアル (別冊インターフェース)』 CQ出版

[28] 『トランジスタ技術 SPECIAL No.27 ハードディスクと SCSI 活用技術のすべて』 CQ出版

[29] 『ポータブル C プログラミング (PORTBLE C SOFTWARE)』 Mark R.Horton 著，長尾高弘訳，Prentice Hall, Inc., Englewood Cliffs, New Jersy，トッパン

[30] 『プログラミング言語 AWK — The AWK Programming Language』 A.V. エイホ/B.W. カーニハン/P.J. ワインバーガー著，足立高徳訳，AT&T Bell Laboratories，トッパン

[31] 『SED & AWK プログラミング — SED & AWK』 Dale Dougherty 著，福崎俊博訳，O'Reilly & Associates, Inc.，アスキー

[32] 『Microsoft C/C++ ランタイムライブラリリファレンス Version 7』，マイクロソフト監修，アスキー

# 索　引

A

<a_out.h> 25
_AES_SOURCE 25
alloca() 25, 79, 84, 85
ANSI/IEEE Std 754–1985 51
ap 90
asctime() 69, 70
auto 型 77, 84, 95

B

brk() 74

C

C ロケール 22
Call by reference 84
Call by value 84
char 型 22, 83
char *型 79, 83, 89
chmod() 47
closedir() 44
__const 型 28, 95
const 型 28
__cpluscplus 28
ctime() 69, 70
<ctype.h> 29

D

difftime() 63
DIR 構造体 44, 45
__DIRECT_FLOAT__ 56, 57
__DIRECT_FPU__ 56, 57
__DIRECT_IOFPU__ 56, 57
dirent 構造体 44, 45
<dirent.h> 25, 44
DOS エラーコード 114
_dos_files() 45
_dos_print() 95
_dos_s_process() 129
double 型 22, 51, 52, 58, 84, 90

E

EDOM 56
_enargv() 99
ENOMEM 74
ERANGE 56
errno 22, 55, 56, 74, 101
<errno.h> 101
execlp() 103
extern 型 77, 95

F

FCB の構造 123
<fcntl.h> 25
float 型 22, 51, 52, 90
foo() 83, 84
free() 72, 82
FSF 28
fstat() 42
ftime() 62
_fullpath() 40

G

getegid() 104
geteuid() 104
getgid() 104
getgrnam() 105
getlogin() 103
getpwnam() 105
getuid() 104
gmtime() 68, 69
GNU プロジェクト 28

## H


HUGE ··· 56
HUGE_VAL ··· 56
**Human68k 例外** ··· 92


## I


i-node ··· 46
IEEE ··· 51
ino ··· 46
int 型 ··· 89, 93, 96
_is68881() ··· 60


## J


<jctype.h> ··· 23, 29
<jstring.h> ··· 23


## L


<limits.h> ··· 25
localtime() ··· 68, 69
long 型 ··· 22, 61, 63
long double 型 ··· 22, 51
longjmp() ··· 110
lstat() ··· 42, 46, 106


## M


main() ··· 98–100
malloc() ··· 72, 74, 77, 79–82, 84
<math.h> ··· 54, 56
matherr() ··· 55
<mbctype.h> ··· 23, 29
<mbstring.h> ··· 23
MC68000 ··· 57
MC68881 ··· 57
MC68882 ··· 57
mktime() ··· 62, 67, 68


## N


new_area ··· 78
NIFTY-Serve ··· 17
NIH ··· 28
NULL ··· 80


## O


opendir() ··· 44


## P


PDS ··· 15
_POSIX_SOURCE ··· 24
printf() ··· 88, 90, 95, 108
puts() ··· 95


## Q


readdir() ··· 44
realloc() ··· 81, 82
rewinddir() ··· 44


## S


S_IEXBIT ··· 47
S_IFBLK ··· 47
S_IFCHR ··· 47
S_IFDIR ··· 47
S_IFIFO ··· 47
S_IFLNK ··· 47
S_IFREG ··· 47
S_IFSOCK ··· 47
S_IFVOL ··· 47
S_IHIDDEN ··· 47
S_IRDONLY ··· 47, 48
S_IRGRP ··· 47
S_IROTH ··· 47
S_IRUSR ··· 47
S_ISGID ··· 47
S_ISUID ··· 47
S_ISVTX ··· 47
S_ISYS ··· 47
S_IWGRP ··· 47
S_IWOTH ··· 47
S_IWUSR ··· 47
S_IXGRP ··· 47
S_IXOTH ··· 47
S_IXUSR ··· 47
sbrk() ··· 72, 74
SCSI エラーコード ··· 115
seekdir() ··· 44
SETBLOCK ··· 72
setjmp() ··· 110
setlocale() ··· 40
SHARP USER'S フォーラム ··· 17
short 型 ··· 22, 83
sig_atomic_t 型 ··· 96
signal() ··· 93
<signal.h> ··· 25

spawnlp() ···················· 103
_stacksiz ···················· 87
stat() ·········· 26, 42–44, 47, 106
stat 構造体 ···················· 43
static 型 ···················· 77, 95
<stdarg.h> ···················· 88
stdio ライブラリ ·········· 22, 25, 29
<stdlib.h> ···················· 25
strcoll() ···················· 40
<stream.h> ···················· 28
strftime() ················ 70, 71
<string.h> ···················· 25
<sys/dos.h> ···················· 23
<sys/iocs.h> ················ 23
<sys/stat.h> ················ 42
system() ················ 74, 105

**T**

telldir() ···················· 44
tempnam() ···················· 103
time() ···················· 62
time_t 型 ················ 61–63, 67
<time.h> ···················· 61
tm 構造体 ················ 62, 67–69
tmpnam() ···················· 103
tzset() ···················· 67

**U**

<unistd.h> ···················· 23

**V**

va_arg ···················· 89, 90
va_start ···················· 89
<varargs.h> ················ 88
void 型 ···················· 93
void *型 ···················· 79
__volatile 型 ················ 28
volatile 型 ···················· 96
volatile int 型 ················ 96

**W**

<wctype.h> ···················· 29

**X**

X3J11 委員会 ···················· 21
_XOPEN_SOURCE ················ 24
X 形式実行ファイル ············ 130

**Z**

Z 形式実行ファイル ············ 130

**あ行**

値渡し ···················· 84
アラインメント ············ 79, 83
インデックスノード ············ 46
閏秒 ···················· 64
エポック ···················· 61
エポックタイム ················ 61
エラーコード ················ 101

**か行**

拡張倍精度 ···················· 58
可変長引数 ············ 22, 24, 88
可変長引数リスト ············ 88
カリフォルニア大学 ············ 22
カレンダータイム ············ 61
カレントディレクトリテーブル ···· 127
環境変数 ···················· 103
関数属性 ···················· 28
起動オプション ················ 98
キャスト ···················· 79
協定世界時 ················ 61, 64
クラスライブラリ ············ 28
グリニッジ標準時 ············ 64
グループファイル ············ 24
グローバルコンストラクタ ····· 32, 99
グローバルデストラクタ ······· 32, 99
原子時 ···················· 64
固定小数点 ···················· 51
コマンドシェル ············ 27, 38
コンパイラドライバ ············ 31

**さ行**

シェアウェア ················ 16
時間帯 ···················· 64
シグナル ···················· 92
シグナル機構 ················ 22
シグナルハンドラ ············ 92, 93
システムコール ················ 18
実行コスト ···················· 78
詳細時間 ················ 62, 67
シンボリックリンク ·········· 42, 106
数値演算コプロセッサ ············ 20
数値演算プロセッサボード ········ 36

スコープ……75, 77, 84
スタック……83
スタックオーバーフロー……86, 100
スタックチェック……86
スタックポインタ……100
スタック領域……86
スーパーバイザモード……35, 98
正規化……52
世界時……64
セシウム原子……64
選択マクロ……28
ソフトウェア例外……92

## た行

大域ジャンプ……22
タイムゾーン……64
地域時間……61, 64
ディスクマップ……48
ディレクトリエントリ……24, 44, 48
ディレクトリストリーム……44
デノーマル数……52
動的メモリ確保……75
ドライブ配置テーブル……129

## な行

夏時間……64
日本時間……64

## は行

配列……75
バークレー校……22
バージョンアップ……17
パスワードファイル……24
ハードウェア例外……93
非正規化数……52
ヒープ領域……72, 86
ファイルアトリビュート……47
ファイルステータス……42
ファイルハンドル……24, 42
ファイルモード……42, 47
副作用……95
副作用のない関数……28
符号つき0……53
物理デバイス情報テーブル……128
浮動小数点……51
フリー……14
フリーウェア……16, 106
プロトタイプ……28
米国規格協会……21
米国電気電子学会……21, 51
米国電信電話会社……21
ポインタ渡し……84

## ま行

マルチバイト……33
未サポート関数……40
無限大……53
メモリ不足……74
メモリブロックの分断……74
戻ってこない関数……28

## や行

ユーザモード……35, 98

## ら行

ライブラリオプション……98
リエントラント……95
暦時間……61, 67
ロケール……22, 29, 40

## わ行

ワークステーション館……17

# あとがき

**こ**ともあろうか執筆期間中にバイクに乗っていたら，11トントラックに“スコーン”とぶつけられ (私は被害者なのねん)，さらに対向車線の乗用車とも大当たりして (宝クジでも買っときゃよかった…)，右手の指2本と右足を骨折して入院していたのは，なにを隠そうこの私です (環状7号線をご通行中だった皆様，大変ご迷惑おかけしました)。

そのため，入院，リハビリと，かなりの期間を費やしてしまい，当初の予定より，ライブラリの作成や原稿が遅れてしまいましたが …，なんとか書き終えることができました (いやぁ，さすがに右手が使えないとつらいわぁ～)。

ライブラリのほうですが，ベースバージョンとしては完成しましたが，高速化や関数追加などは，まだまだパソコン通信上でサポートを行うつもりでいますので，レポートなどお待ちしております。また，関数自体は厳しくチェックしたつもりではありますが，なにぶん，関数の数が多いためバグの潜んでいる確率も多ございますので …こちらのほうもなにとぞよろしくお願い致します。

ちなみに，この本の副題は「サルでも書けるライブラリ」です。

ウゴウゴルーガ生放送のとろりんの「ウゴルーてんきよほう」を見つつ…

―――――――― 大西　恵司

**ま**ったく，とんでもないプロジェクトに首をつっこんでしまったものだと思う。

私が最初にライブラリの作成を始めたのは，あるネットのオフ会でもらってきたGNUソフトの移植をするためだった。こうした**UNIX**系のソフトをいわゆる**MS–DOS**系に移植するときには，必ずといっていいほど**UNIX**側のライブラリを何とかしなくてはならない。そのため，「どうせ作るならきちんとした仕様にあわせよう」と***AES/OS***の仕様書を買い込んで，ポツリポツリと必要な関数だけ形成していった。

そんなおり，村上氏がNIFTY–Serveでフリーなライブラリを作ろうとしているという話を耳にした。そのころは，まだこのような書籍という形ではなくネッ

トワーク上で有志が集まって，皆が作ったライブラリを持ち寄るというものだった (本文にこのあたりの経緯が述べられている)。そこに私も参加させてもらい，今回の書籍化のお手伝いとなった。

最初は，いままで有志の方々から寄せられたライブラリを基にしていたのだが，仕様統一や実用上の問題などを考慮して，一般のプログラマが実用に耐え得るであろう関数仕様としていった。もっとも，作成中にもコンピュータ業界はめまぐるしく変化していったので，当時先進だったものが，今は当たり前のものとなってしまった。その意味では，従来の純正ライブラリに不満を感じていた方々(または，純正の守備範囲では足りなかった方) には満足していただけると思う。

今回の書籍化にあたり，企画を取りまとめられた村上氏，ソフトバンクの方々にはいろいろとご迷惑をかけたが，辛抱強くサポートしていただき，今こうしてあとがきを記すことができた。この場を借りて感謝したい。

また，私にきっかけを投げかけ，数々の助言を与えてくれた方々(誰とはいわないが) にも感謝の言葉を送りたいと思う。

――――――――――――――― 萩野　祐二

結局，この本を完成させるのに 1 年もかかってしまいました (もっとも，ずっとかかりっきりだったわけではありませんが ···)。本当はもっと早く完成する予定だったのが，いつの間にやらこのありさま。編集の方々にもいろいろと面倒ばかりかけてしまって，まったくもってお恥ずかしいかぎりです。まだまだやり残したことは多いのですが，とりあえずここでひと区切りとします。

しかし，なんですね。人間，寝なくても結構大丈夫なもんですね (「おもいっきし寝とるやんけ」なんてつっこまないように)。おかげさまで，この 1 年の間に，私は寝なくても倒れない方法を体得することができました。もっとも，かの翁にいわせれば，まだまだ「寝すぎ」なんでしょうが ...。

そんなことはともかく，**X68030** について。なにしろ，**X68030** が発売されたのがちょうど最後の追い込み時期だったので，かの翁はともかく，私自身はあまり **X68030** についてよく調べていません。187cm さんから **X68030** をお借りして一応動くようにはしたものの，**X68030** に対する *LIBC*の対応は，はっきりいってまだ満足できるものではありません (**X68030** というよりも **Human68k** が ver.3 になったことが大きいのよ ··· 結局のところ)。そこのところは，これからということで大目に見てやってください。

と，いうわけで皆さん，*Have a nice hacking!!* (訳：「バグがあったら直してちょうだい」)

――――――――――――――― 村上　敬一郎

最後に，デバッグやレポートをしてくれた皆さん，どうもありがとうございました。特に次の方々にはたいへんお世話になりましたので，この場を借りてお礼申し上げます。

Abechan，板垣さん，187cm，沖@沖さん，TomY さん，ちゃぷにさん，homy さん，Bun. さん，Mad Player さん，真里子さん，PEKIN-NET の皆さん，そして謎の Mailing List x6user の皆さん。

また，**Human68k** の解析は PEKIN–NET にアップロードされていた，謎の“書籍が出ないのなら自分で書いてしまえプロジェクト”さんの書いた解析データを基に作成しました。このデータは，ライブラリ作成時にもたいへん参考になりました。どうもありがとうございました。

ISBN4-89052-534-3

C0055 P6300E

2冊セット定価6,300円
（セット本体6,117円　分売不可）